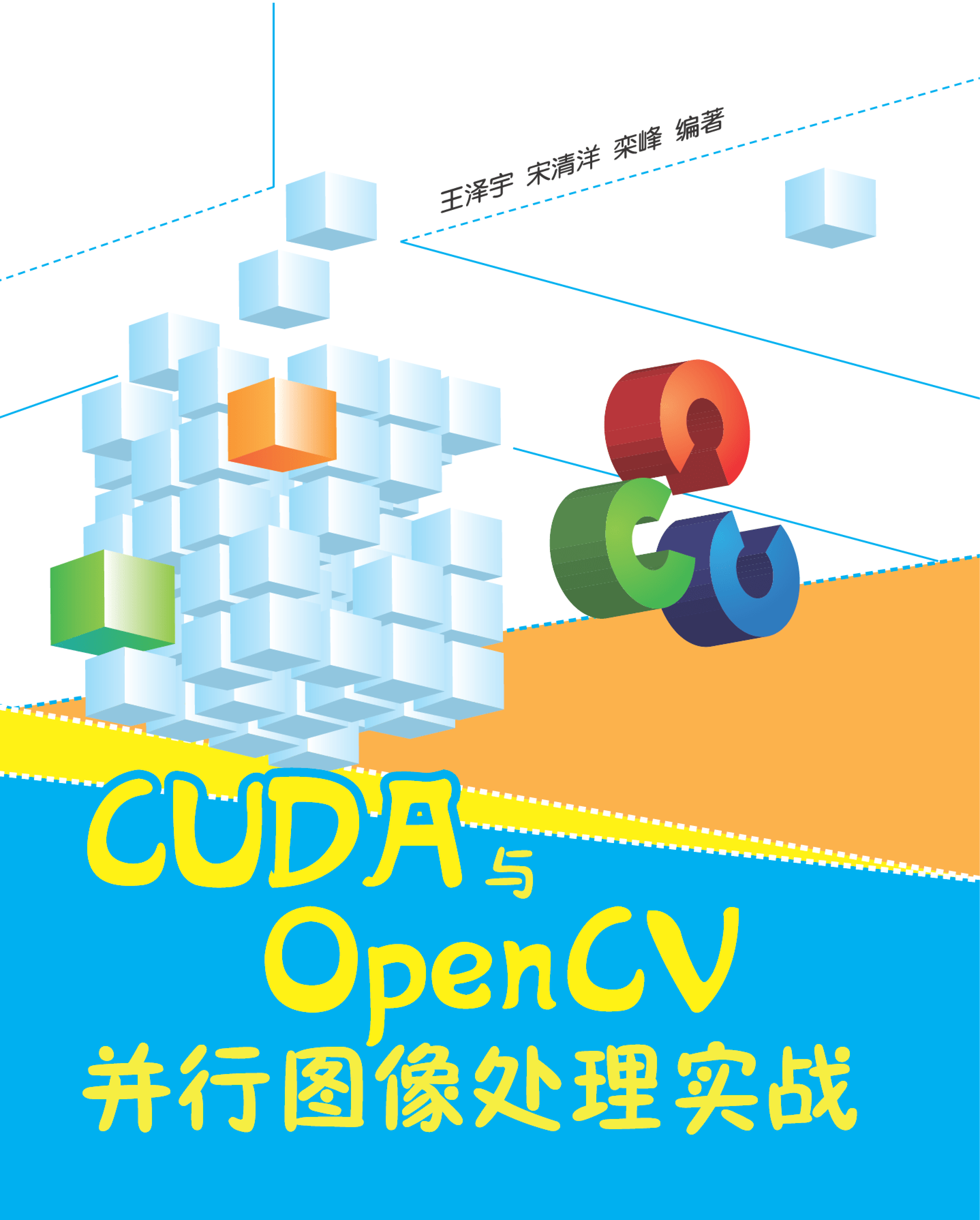


清华大学出版社

CUDA 与 OpenCV 并行图像处理实战

王泽宇　宋清洋　栾　峰　编著

清华大学出版社
北　京

内 容 简 介

本书主要介绍图像处理和 GPU 加速的基本原理、主要技术和典型应用。全书共分为 5 章，详细介绍了 OpenCV 的环境搭建，OpenCV 在图像处理算法中的应用，OpenCV 如何与 CUDA 进行编译，以及如何使用编译后的 OpenCV 库驱动 GPU 加速传统的图像处理算法。

本书可作为信号处理、通信工程、计算机应用、广播电视、自动控制、生物医学工程、地理信息等领域的工程技术人员，以及大专、本科院校相关专业的高年级学生研究图像处理技术的参考书。

图书在版编目(CIP)数据

CUDA 与 OpenCV 并行图像处理实战/王泽宇，宋清洋，栾峰编著. —北京：清华大学出版社，2019
ISBN 978-7-302-51048-2

Ⅰ. ①C… Ⅱ. ①王… ②宋… ③栾… Ⅲ. ①图像处理软件—程序设计 Ⅳ. ①TP391.413

中国版本图书馆 CIP 数据核字(2018)第 192050 号

责任编辑：赵 凯 李 晔
封面设计：常雪影
责任校对：梁 毅
责任印制：沈 露

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦 A 座 **邮 编：**100084
社 总 机：010-62770175 **邮 购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：北京密云胶印厂
经 销：全国新华书店
开 本：185mm×260mm **印 张：**17.5 **字 数：**422 千字
版 次：2019 年 6 月第 1 版 **印 次：**2019 年 6 月第1 次印刷
定 价：69.00 元

产品编号：072020-01

序言

数字图像处理是实现计算机视觉的关键技术。在当前实际应用中，计算机视觉系统对实时性和准确率的要求越来越高，需要处理的数据量越来越多，涉及的计算量也越来越大，这使得目前个人计算机的数据处理能力不能满足实际需求。但随着图形处理器(Graphic Processing Unit，GPU)的急速发展，使用 GPU 进行加速计算通常能获得处理速度的大幅提升，因此使用 GPU 建立实时、准确、高效的计算机视觉系统成为必然趋势和研究热点。

然而，目前使用 GPU 做图像处理的相关中文资料普遍存在不足：

第一，资料较少且信息零散，不成系统。通常是这个博客中讲解几个技术点，那个论文中介绍几种算法，缺少完整的、条理清晰的归纳。

第二，较为复杂的环境搭建让很多初学者望而却步。

本书是一种非常适合初次接触并行图像处理技术的本科生和研究生的入门级读物。书中，首先以 OpenCV 和 CUDA 的基础知识为出发点，介绍并行处理数据的工作原理；然后详细介绍 OpenCV 和 CUDA 的环境搭建过程和其中可能遇到的问题及解决方案，帮助读者顺利完成开发环境搭建；最后通过丰富的算法和例程，由浅入深地介绍多个并行图像处理算法，为读者未来更深入地研究并行图像处理技术提供实践基础。

我由衷地希望有志加入数字图像处理技术开发的莘莘学子，能凭借此书更快速地打好基础，通过对 GPU 的并行运算能力的理解与应用，开启图像处理技术研发的大门。

隆克平

2019 年 3 月于北京科技大学

前言

随着大数据时代的来临，越来越多的图像需要实时处理。随之而来的使用 C++编程的机器视觉库 OpenCV 以及驱动 GPU 的 CUDA 也变得越来越火热。

OpenCV 是机器视觉领域非常著名的开源库，它几乎被应用到机器视觉的所有领域，其功能几乎涵盖每个研究方向。OpenCV 包含了底层的图像处理、中层的图像分析以及高层的视觉技术。而且，其算法紧跟视觉前沿，将最新的算法纳入其中。特别是 OpenCV 2 系列的出现，可以使用 C++进行编程，并且可以使用 GPU 为图像处理进行加速。OpenCV 在图像界是相当重要的工具，也是很多图像领域研究人员极力推荐的库。

CUDA 作为一种并行计算架构，是以 GPU 为数据并行计算设备的软硬件体系。CUDA 以 C 语言为基础，可以直接用 C 语言写出在显示芯片上执行的程序，而不需要去学习特定的显示芯片的指令或特殊的结构。因此，CUDA 被广泛应用在视频编解码、金融、地质勘探、科学计算等领域。

作为并行图像处理的入门级教材，本书将并行计算架构 CUDA 和机器视觉库 OpenCV 结合，以大量示例程序为主线，详细介绍了如何搭建 OpenCV 环境，如何使用 Cmake 编译 CUDA 和 OpenCV，以及环境搭建过程中可能出现的错误和解决方案。编写本书的初衷是希望更多初步接触 GPU 和图像处理的读者可以快速搭建好环境并快速了解 OpenCV 和 CUDA 的基础知识，节省入门消耗的时间。

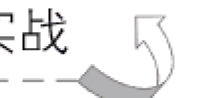
由衷感谢我的导师宋清洋对于我学业和生活上的支持与鼓励，以及对这本书的付出。感谢栾峰老师对我学业上的指点，没有他的指点也就不会有这本书的诞生。感谢我的好兄弟郑建斌和学姐包锡伟在我学习图像处理的过程中对我的指导。

真心希望读者可以轻松地入门并行图像处理技术。由于作者水平有限，书中难免有不足之处，恳请读者批评指正。

王泽宇

2019 年 3 月于东北大学

目录



第1章 并行图像处理概述

1.1 计算机的构成

本节将从计算机的组成部分来普及计算机的硬件知识，将介绍计算机的中央处理器、主板等硬件，并详细介绍本书后面并行处理所基于的硬件——GPU(Graphics Processing Unit)。

1.1.1 计算机硬件构成

计算机的硬件系统可以按照工作原理和实际应用两种方式进行分类。

按照工作原理可以分为五大部分：运算器、控制器、存储器、输入设备和输出设备。运算器和控制器合称中央处理器，也就是CPU(Central Processing Unit)，它是计算机工作的核心。存储器包括内存和外存，内存用来存储运行中的临时数据；外存用于存储应用程序和用户数据，硬盘光盘等都属于外存。输入设备用于给计算机输入程序或数据，如键盘、鼠标、扫描仪等。输出设备是将计算机处理后的结果送到外部设备，如显示器、打印机等。

从实际应用分为两方面：内部设备和外部设备。内部设备分为CPU、内存、主板、显卡、声卡、网卡、光驱、电源。外部设备分为显示器、键盘、鼠标、音箱、耳机、打印机、扫描仪、手写板等与计算机相关的设备。

中央处理器，即CPU，是一块超大规模的集成电路，是一台计算机的运算核心(Core)和控制核心(Control Unit)，它的主要功能是解释计算机指令以及处理计算机软件中的数据，CPU的处理方式是传统的串行处理，如图1-1所示为CPU的外观。



图1-1 CPU外观图

显卡即GPU，又被称作图形处理器、显示核心、视觉处理器、显示芯片，是一种专门在计算机、工作站、游戏机和一些移动设备(如平板电脑、智能手机等)上进行图像运算工作的微处理器，GPU的内部结构与CPU不同，它处理数据的方式是并行处理。在多数情况下，显卡指的是包含了GPU芯片和辅助芯片的电路、风扇、接口等部分的一个完整的、可以直接使用的硬件设备，而GPU指的是做图像处理计算的GPU芯片。显卡的外

貌如图 1-2 所示，显卡内部的 GPU 芯片如图 1-3 所示。



图 1-2　计算机显卡

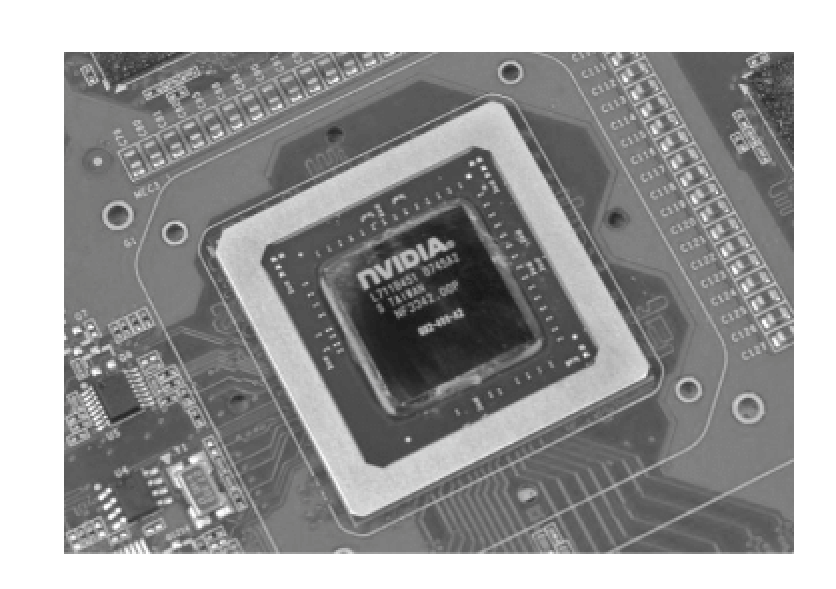


图 1-3　显卡内的 GPU 芯片

内存是计算机运行程序的地方。通常使用计算机运行的所有程序都是在内存中执行的，因此内存容量的大小对计算机的运行速度影响也比较大。计算机使用久了，运行出现卡顿现象，多数原因是内存容量不够大。内存所处的硬件被称作内存条，内存条的外观如图 1-4 和图 1-5 所示。在内存条生产领域，较为知名的企业有三星、金士顿、英飞凌、现代、Smart 等。近几年，这些内存条生产商逐步采用了小型的内存条来代替传统的大内存条，俗称小卡和大卡。小卡与大卡的区别主要体现在小卡宽度比大卡窄，小卡也可以插在原本大卡的接口上，不过两边没有扣子卡住小卡。小卡的优势是体积小，给其他硬件提供了更大的空间，不过其劣势就是集成度太高，散热等性能不如大卡，同样内存容量的大卡和小卡，从稳定性角度来看，大卡更胜一筹。

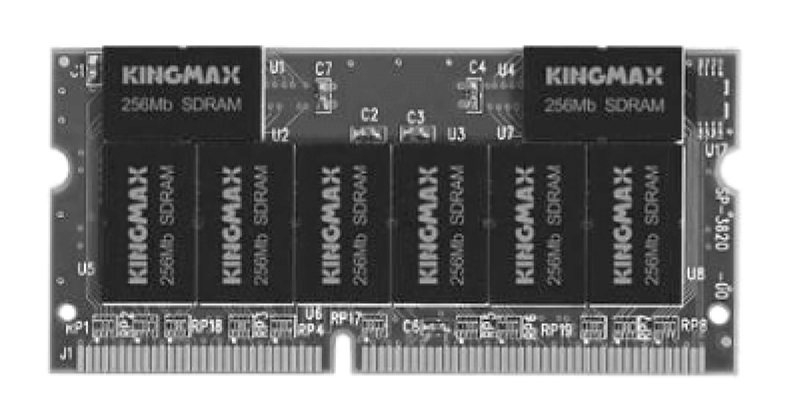


图 1-4　笔记本电脑中的内存条

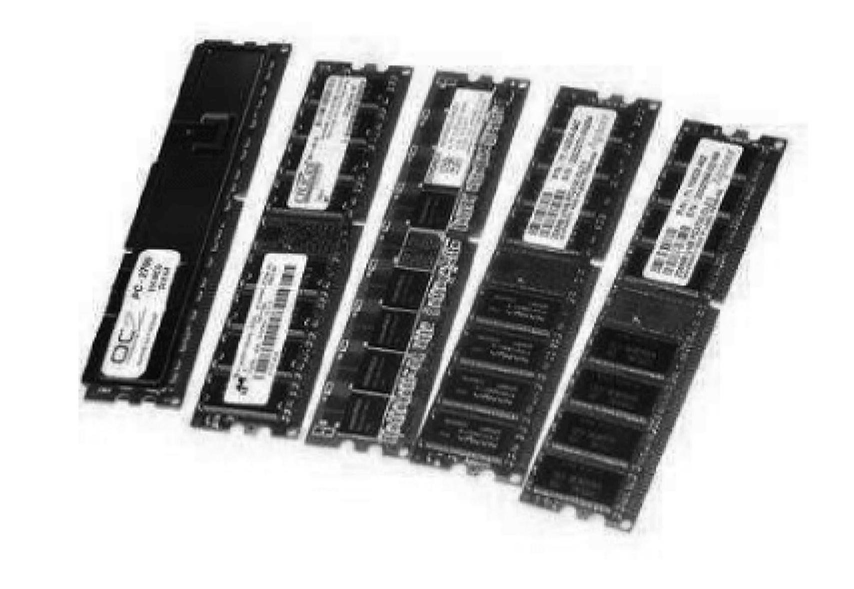


图 1-5　台式机中的老式内存条(大卡)

硬盘作为构成计算机硬件系统的存储设备，具有非常重要的地位。可以说，没有硬盘，计算机就无法正常工作。硬盘集机、电、磁于一体，结构相当复杂，图 1-6 和图 1-7 分别为硬盘的外部结构和内部结构。硬盘主要分成固态硬盘(Solid State Drive，SSD)、传统硬盘(Hard Disk Drive，HDD)和混合硬盘(Hybrid Hard Drive，HHD)三大类。固态硬盘是使用固态电子存储芯片阵列制成的硬盘，由控制单元和存储单元(Flash 芯片、DRAM 芯片)组成。固态硬盘的优点是：读写速度快；抗摔性好；低功耗；无噪声；工作温度范围大；轻便。缺点是：容量小；寿命有限；售价高。传统硬盘就是常说的机械硬盘。机械硬盘



图 1-6　硬盘的外部结构

使用了传统的工艺，虽然读写速度和稳定性不如 SSD，但是因为造价成本低，所以仍然占有很大的市场。混合硬盘是将 SSD 和 HDD 两种硬盘混合在一起，它既包含了 HHD 的大容量，又有 SSD 的闪存模块，目前市面上常用的笔记本电脑都是使用这种硬盘。

主板，又叫主机板、系统板或母板。它安装在机箱内，是计算机最基本的也是最重要的部件之一。主板一般为矩形电路板，上面安装了组成计算机的主要电路系统，一般有 BIOS 芯片、I/O 控制芯片、键和面板控制开关接口、指示灯插接件、扩充插槽、主板及插卡的直流电源供电接插件，通常情况下还会焊接上声卡、网卡等装置，CPU 作为处理数据的核心，也会焊接在主板上。其外观如图 1-8 所示。

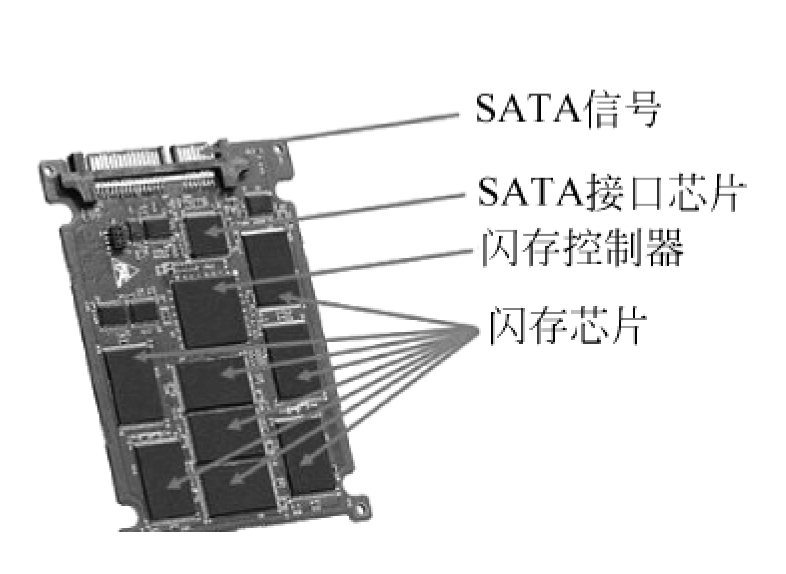


图 1-7　固态硬盘的内部结构

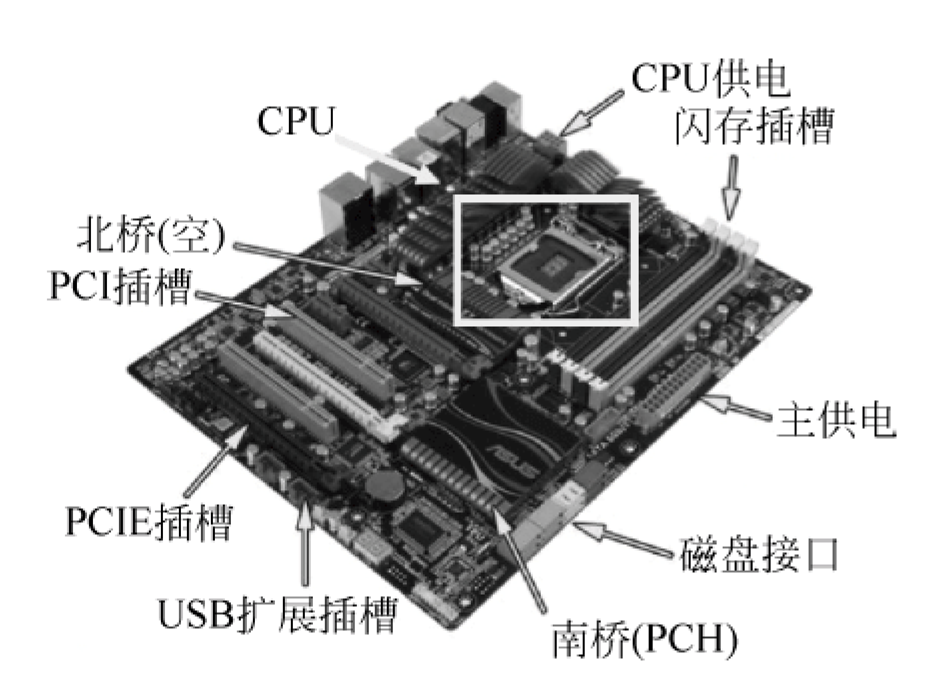


图 1-8　主板外观

1.1.2　显卡和 GPU

1.1.1 节已经对计算机中的显卡做了简单介绍，本节将针对后面并行处理需要用的硬件平台显卡和 GPU 做详细的介绍。

1. 显卡的分类

显卡的全称是显示接口卡，是计算机最基本的配置之一。显卡作为计算机的重要组成部分，承担输出显示图形的任务。用途是将计算机系统所需要的显示信息进行转换驱动，并向显示器提供信号，控制显示器的正确显示。没有显卡，计算机中的图像就无法处理，更无法在显示器上输出。

显卡目前分两种——集成显卡和独立显卡。

独立显卡简称为独显，是一个独立于主板之外的硬件，独立显卡具备单独的显存，不占用系统内存(但是独立显存不够用时可以共享内存为显存)，技术上优于集成显卡。独显由于拥有独立的一套运行环境，这使得其核心运算有很大的发挥空间，因此相对于集成显卡有更好的性能。如图 1-9 所示为独立显卡的配套硬件及其 GPU 芯片。

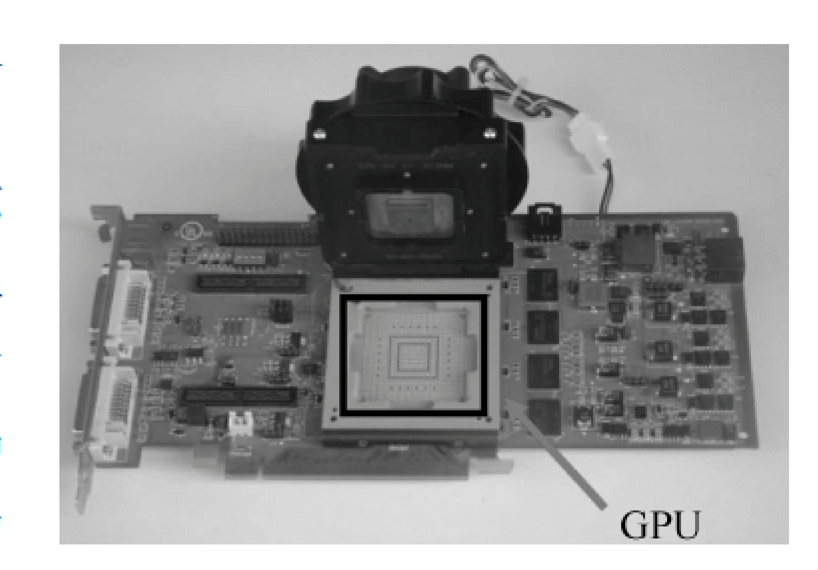


图 1-9　独立显卡 GPU 芯片

集成显卡简称集显，不带有显存，性能一般都比不过同等级的独立显卡。集成显卡分为两种：一种是焊接在主板的北桥中；另外一种是封装在 CPU 中，与 CPU 放在一起工作，被称为核心显卡，简称核显。两种不同的集成显卡因为没有独立的显存，所以工作时的显存都是从内存中分享而来，当运行较大型的程序或

游戏时，会占用相当一部分内存。近些年核显的性能得到了巨大的飞跃，核心显卡对内存的依赖越来越严重，在一定程度上也会影响 CPU 的性能。如图 1-10 和图 1-11 所示为计算机中焊接在北桥中的集成显卡。

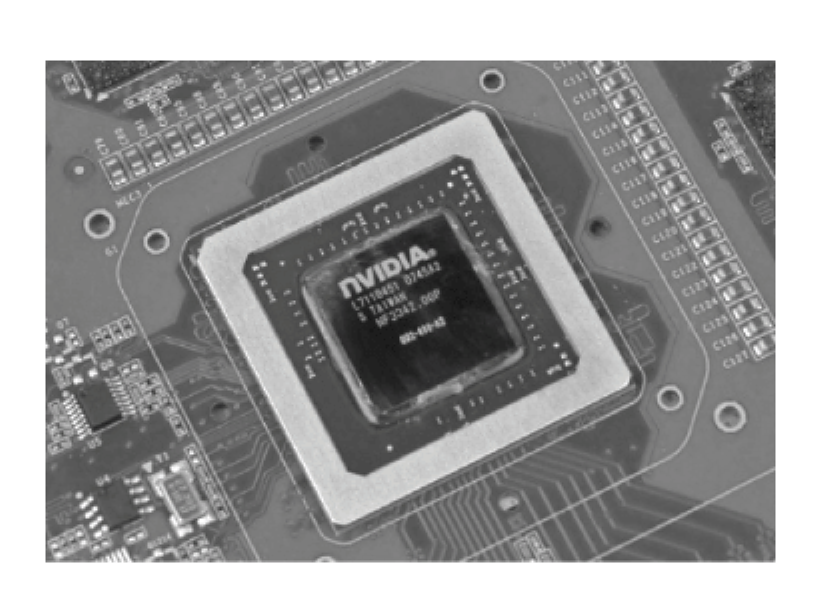


图 1-10　集成显卡

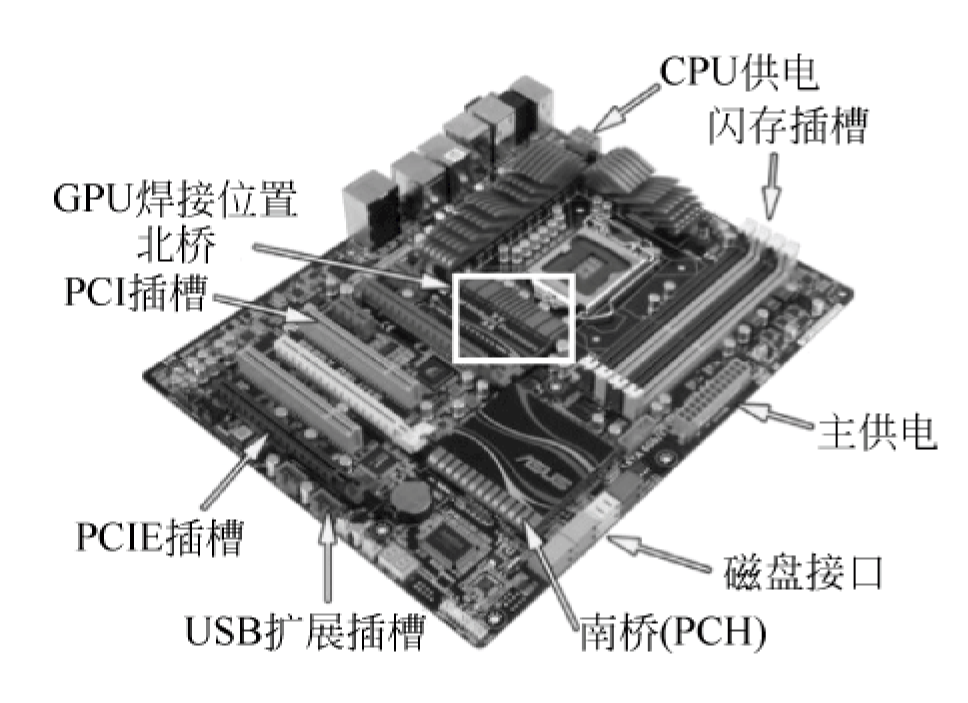


图 1-11　主板上的集成显卡

现在的台式机和笔记本电脑中通常都同时拥有独立显卡和集成显卡，有一些会智能地在显示简单图像时自动使用集成显卡降低功耗，在需要高清画质等时候自动使用独立显卡提高画面效果。

2. NVIDIA 显卡

显卡的厂商主要有英伟达(NVIDIA)和 AMD 两家公司，现在的 AMD 显卡是原本制作 GPU 的 ATI 公司，后被 AMD 公司收购合并成一家，这两家的显卡俗称为 N 卡(NVIDIA 公司的显卡)和 A 卡(AMD 公司的显卡)。虽然苹果、Intel 等公司也在开发显卡，但市场份额远不如这两家。因为本书后续用到的 CUDA 都是使用了 NVIDIA 公司的独立显卡，所以接下来将对目前市面上常见的 NVIDIA 公司的独立显卡进行介绍。

目前 NVIDIA 公司的独立显卡主要分成三大类：Tesla 系列、Quadro 系列和 GeForce 系列。Tesla 系列显卡是专用的服务器级别的显卡，常用于大规模并行计算，超长时间的连续运行也不会大幅降低运行速度，非常适用于机器学习，但是一般价格很高昂，代表显卡有 Tesla K40 和 Tesla K80，如图 1-12 所示为 Tesla 显卡的外形。有人将 Quadro 显卡也称为 Q 卡，Q 系列的显卡是专业的图形设计显卡，在制图方面被广泛使用。其价格低于 Tesla 显卡，长时间持续工作的续航能力也低于 Tesla 显卡，如图 1-13 所示为 Quadro 显卡。GeForce 系列就是市面上最常见的游戏显卡。GeForce 系列的中文名称是“精视”，主要面对大众用户，用途主要为娱乐，支持市面上绝大多数的 3D 游戏，也是 NVIDIA 公司销售量最高的类型，到现在为止的最新版为 GTX 1080Ti。相对于专业显卡，GeForce 系列相对便宜，而且支持 CUDA、cuDNN，所以现在做深度学习一般也会使用 GTX 1080Ti。市面上常见的 GeForce 系列显卡基本都是 GTX 系列。如前面提到的 GTX 1080Ti 和 GTX 1080。其中 GTX 是定位性能级，从前有 GT 和 GTX 两大类，GT 的全称为 Graphics Processor protoType，定位为次高端显卡；GTX 全称为 Graphics Processor prototype eXtreme，定位为高端显卡。随着显卡技术的进步，GT 系列已经渐渐为低端显卡，而现在市面上常见的 GeForce 系列显卡也都是 GTX 开头的显卡。GTX 显卡后的编号，以 GTX 1080Ti 为例，前两位代表第 10 代显卡，后两位代表性能，其实本质上是不同的架构和工艺，有没有 Ti 代表是否为增强版。从性能角度，GTX 1080Ti 优于 GTX 1080，GTX 1080 优于 GTX 1070，

GTX 1080 优于 GTX 980。



图 1-12　Tesla 显卡



图 1-13　Quadro 显卡

以上介绍的都是台式机的独立显卡和集成显卡，笔记本电脑因为没有较大的空间，所以其独立显卡也制作得非常小巧，并且焊接在主板上，如图 1-14 所示为笔记本电脑中的独立显卡。笔记本电脑中的独立显卡没有台式机中独立显卡那么好的风扇、辅助电路等，所以性能远不如同名称台式机的独立显卡，但是这种独立显卡也比普通的集成显卡性能要好，而且这种独立显卡也支持 CUDA 并行计算。

3．性能参数介绍

显卡的显存也称为帧缓存，它的作用是存储显卡芯片处理过或者即将提取的渲染数据，同计算机的内存一样，显存是显卡用来存储要处理的图形信息的部件，如图 1-15 所示。

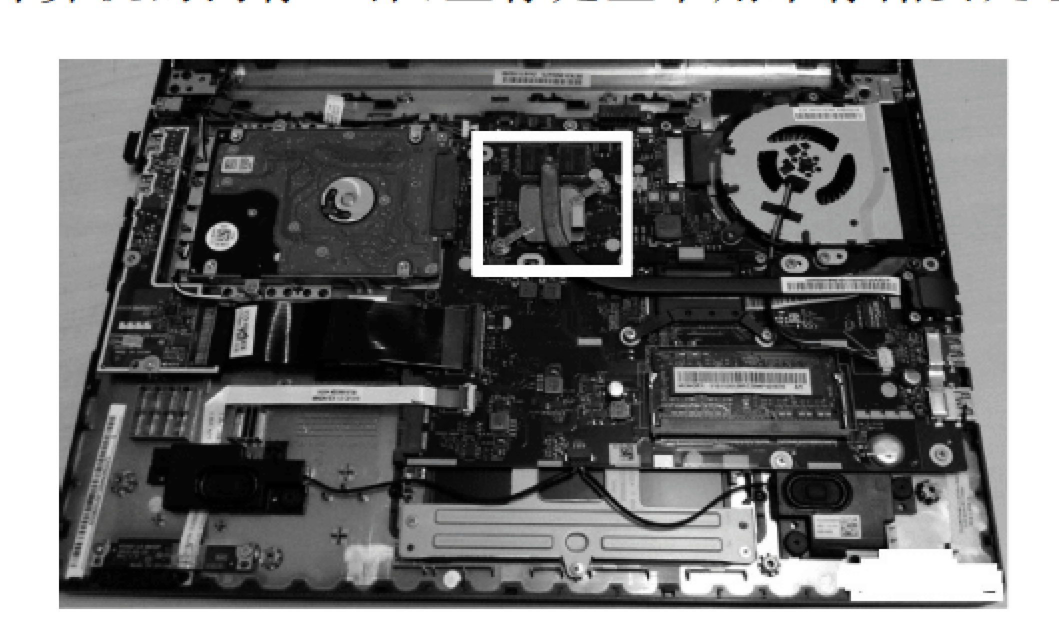


图 1-14　笔记本上的独立显卡

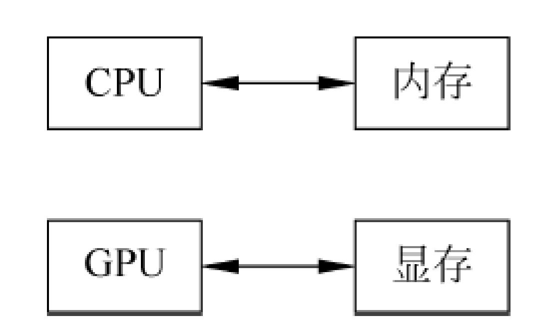


图 1-15　GPU 与显存的关系

显存位宽是显存在一个时钟周期内所能传送数据的位数，位数越大则瞬间所能传输的数据量越大，是显存的好坏的重要参数之一。显存带宽＝显存频率×显存位宽÷8，那么在显存频率相当的情况下，显存位宽决定显存带宽的大小。同样，显存频率为 500MHz 的 128 位和 256 位显存，那么它俩的显存带宽将分别为：128 位显存带宽＝500MHz×128÷8＝8GB/s，而 256 位显存带宽＝500MHz×256÷8＝16GB/s，是 128 位的 2 倍，可见显存位宽在显存数据中的重要性。

核心频率指的是显卡显示核心的工作频率，它在一定程度上反映出了显示核心的性能。可以将其类比为 CPU 的主频，数值越高，性能越好。

显存频率是指默认情况下，该显存在显卡上工作时的频率，以兆赫兹（MHz）为单位。显存频率一定程度上反映着该显存的速度。在一定程度上可以类比为计算机的内存频率。

4．GPU-Z 查看显卡参数

显卡的性能参数一般在购买显卡附赠的说明书上会有详细说明，或者网上也有同型号显卡的性能参数。如果在不确定自己显卡的型号，或者想给显卡做一个详细的测试，则需要

较专业的测试软件——GPU-Z,其 Logo 如图 1-16 所示。

GPU-Z 是一款非常出名的处理器识别工具,软件约占 4Mb,非常轻便[1]。GPU-Z 可以检测如下参数:

① 检测显卡的型号、制作工艺、核心面积、晶体管数量、渲染器数量及生产厂商。

② 检测光栅和着色器处理单元数量及 DirectX 支持版本。

③ 检测 GPU 核心、着色器和显存运行频率,显存类型(生产厂商)。

图 1-16 GPU-Z 的 Logo

④ 检测像素填充率和材质填充率速度。

⑤ 实时监测 GPU 温度、GPU 使用率、显存使用率以及风扇转速等相关信息。

⑥ 检查显卡的插槽类型和显卡所支持的附加功能与显卡驱动信息及系统版本。

主要特性有:

① 支持 NIVIDIA、ATI、AMD 和 Intel 的显卡或图形设备。

② 显示适配器型号和 GPU 型号,并且能显示其具体信息。

③ 显示显卡加速,默认时钟频率以及 3D 时钟频率(如果设备支持)。

④ 软件内包括了一个 GPU 负载测试,来监测显卡运行在 PCI-Express lane 上的状态。

⑤ 结果信息经过验证。

⑥ GPU-Z 可以备份显卡的 BIOS。

⑦ 不需要安装,但支持安装操作。

⑧ 支持 Windows XP/Vista/Windows 7/Windows 8/Windows 10。

如图 1-17 所示为 GPU-Z 检测 GTX 1070 的结果截图,GPU-Z 的一大优势是测量的很多参数都是基于硬件当时的状态,所以造假的 GPU 很难在 GPU-Z 的测试下鱼目混珠。



图 1-17 GPU-Z 检测 GTX 1070 显卡

1.1.3　显卡的发展史

了解显卡的基础知识后，本节将介绍显卡的发展历史。

早期的计算机因为没有图形界面，所以没有考虑到图像显示的问题。随着计算机的发展和图形操作系统的不断开发（最典型的是 Microsoft 公司的 Windows 系列），图像界面已经普及，专门应对图像显示的 GPU 也在此期间得到高速的发展。

最早的个人计算机的图形处理单元是 IBM 公司在 1981 年推出的 CGA（Color Graphics Adapter）和在 1984 年推出的 EGA（Enhanced Graphics Adapter）。EGA 能同时显示 16 色，分辨率可以达到 640×350 像素，帧缓存达到 256KB。

IBM 公司在 1987 年又推出了 VGA（vidio graphics array）。VGA 实现了在单个芯片上包含图形的所有操作，如硬件的平滑滚动、屏幕的分割和扫描等操作。由于 Windows 操作系统的广泛使用和快速发展的趋势，各种 2D 和 3D 的图像处理芯片也相继出现。

当时高档的图形工作站霸占了 CAD 工作站的整个主流市场，硅谷公司（SGI）在 1992 年推出了 OpenGL，使其成为了第一个 2D 和 3D 操作系统的 API。1994 年，Matrox 公司为了发展个人计算机的 CAD 市场，推出了第一个应用于计算机的 3D 图形加速器 Matrox Impression。

1998 年，NVIDIA 公司推出了实时交互视频（Real-time Interactive Video and Animation accelerator，RIVA）和动画加速器（Twin Texel，TNT）。

1999 年，NVIDIA 公司发布了 GeForce 256，它实现了在芯片上集成变换、光照、建材、纹理和染色引擎，同时能够与 OpenGL 1.2 和 DirectX 7.0 兼容，达到了 SGI 的高端专业 3D 工作站的水平。2001 年，NVIDIA 推出的 GeForce3 优化了固定流水线的工作模式，首次提出了顶点着色器和像素着色器的概念，初步实现了部分电路的用户可编程性，其可编辑性使图像效果的实现不再受显卡的固定渲染管线限制，从而使多个顶点和多个像素点流入处理单元，分别通过同一程序进行独立的处理。

NVIDIA 公司的主要竞争对手 ATI（Array Technology Industry inc，曾在图形渲染技术上领先，在 2006 年被 AMD 公司收购）公司在 2002 年推出了第一个符合 DirectX 9.0 规范的加速器。在该加速器中，顶点着色器和像素着色器可以方便、灵活地实现循环和长浮点数的运算。DirectX 9.0 进一步加强了像素着色器和顶点着色器的功能，使原来的汇编级的语言发展成为 C 语言风格的高级语言 HLSL，进而产生了类似于 CPU 的程序编译的概念。在同一时期，OpenGL 也根据用户的需要不断地改进，发展成为 OpenGL 1.5 以及后来的 OpenGL 2.0。并在此过程中形成了类似 C 语言风格的高级染色语言 GLSL。

2006 年，Microsoft 公司推出了 DirectX 10，其试图统一各种染色器，在理论上消除处理的瓶颈。此时，NVIDIA 公司推出了通用并行架构（Compute Unified Device Architecture，CUDA），AMD/ATI 推出了 CAL/CTM 和后来的 Stream SDK 作为 GPU 编程模型的抽象层。这些 GPU 编程框架使用户可以利用 GPU 强大的计算能力进行通用计算，实现 GPGPU 计算。

2010 年，NVIDIA 公司推出的 GPU Fermi 架构集成了大约 30 亿个晶体管，其拥有 512 个 CUDA 核心，存储器接口达到 384 位宽，存储器峰值带宽达到了 230GB/s，其主要应用于实时图形处理和大规模并行计算领域。同时，AMD 采用 40nm 工艺推出了 Radeon 系列，有

20亿晶体管,也转向通用计算和移动图形计算领域的研究。此外,NVIDIA公司的CUDA编程框架和国际媒体处理标准协会KHRONOS推出的并行计算语言标准OpenCL都在很大程度上加速了GPGPU的发展。

2016年,NVIDIA公司推出了显卡GTX 1080和CUDA 7.5,在2017年又推出了GTX1080Ti和CUDA 8.0。

在这段发展史中,需要记住的时间节点是在2010年,在那一年NVIDIA做出了一款很实用的CUDA,这个CUDA已经可以做一些很全面的开发。这个时间点也说明,如果想使用CUDA,尽量使用2010年以后推出的NVIDIA显卡,虽然最近AMD公司推出的新版显卡也可以支持CUDA,但因为架构不同,兼容性还是不如NVIDIA自己的显卡。

图1-18所示为显卡发展的简单历史。

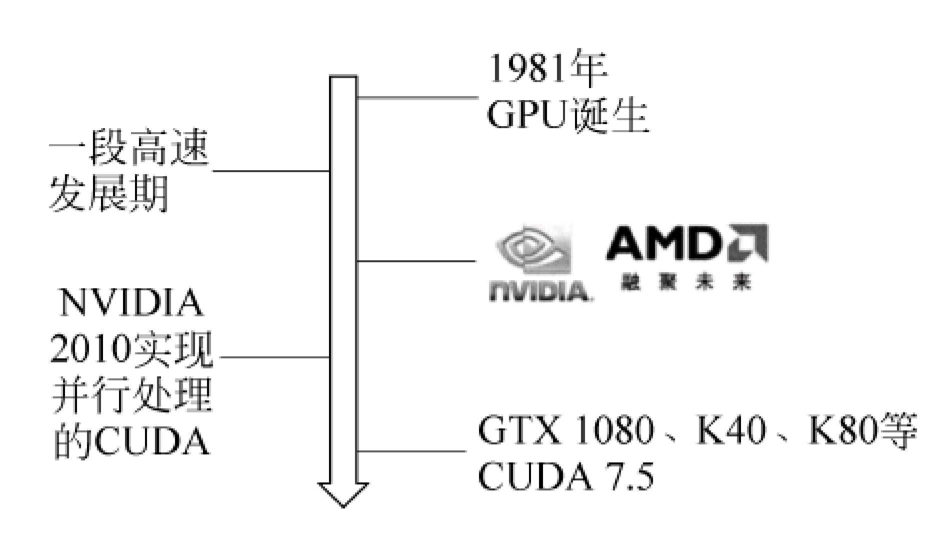


图1-18 显卡发展历史的简略图

1.2 并行计算

并行计算(Parallel Computing)或称平行计算是相对于串行计算来说的。它是一种一次可执行多个指令的算法,目的是提高计算的速度,通过扩大问题求解规模,解决大型而复杂的计算问题。并行计算可分为时间上的并行和空间上的并行。时间上的并行就是指流水线技术,而空间上的并行则是指用多个处理器并发的执行计算。

从广义上来讲,并行计算包括时间上的并行处理和空间上的并行处理,时间上的并行主要指在程序执行多条指令时,重叠进行操作的一种准并行处理实现技术,另外,多任务分时分复用也算是一种并行处理技术。空间上的并行则是用多个单元并发的执行计算,这些处理单元可以是多点式分布的CPU或者GPU,或者是一个GPU内部的许多处理线程,还可以是其他的异构形式。可见,时间上的并行并不能算是严格意义上的并行计算技术,而如何更有效地在空间上并行处理则是现在主流的研究方向[2]。

在并行处理进入公众视野之前,传统的串行处理是数据处理的主流。传统的串行计算就如同是沙漏中的沙子,堆积如山的数据只能排队等着被处理器逐一处理。所谓的多任务并行处理,也只是对不同任务分配不同的时间段来处理,这种时分复用的处理方法,并没有从本质上提高整体运算的速度。为了解决庞大数据量和有限处理能力之间的矛盾,研究者们发明了多核技术甚至是多处理器技术,目的也类似于在大量的沙粒下面多开通一些通道,让沙粒更快地流下,如图1-19左图所示,这也是

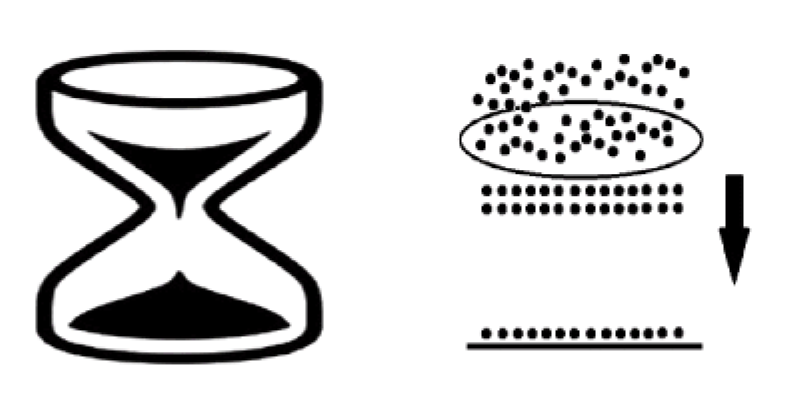


图1-19 并行处理示意图

早期的并行计算的雏形。随着硬件和软件技术的不断进步,时至今日,如果仍用沙粒来比喻数据,那么并行计算技术已经成为一个大大的筛网,能够很快地让这些数据像沙粒通过筛网一般并行处理,如图 1-19 右图所示。每一个筛孔就是一个处理器的一个单元,将无数个单元准备在同一个平面中,每个单元都是并行的,相互之间没有任何影响,这样就可以对所有数据同时进行处理。虽然每个处理单元的速度可能并不是特别快,但是当处理单元数目巨大时,就可以使整体的计算时间大大缩减,相比之下,传统的串行处理就远远不及并行处理的速度了[3]。

在这个数据量呈指数级暴涨的时代,越来越多的数据需要处理,对于一些对计算时间有严格要求的领域甚至需要对数据进行实时处理。即便传统的 CPU 计算能力异常强大,但是在巨大的数据量面前仍然显得不堪一击;与此同时,CPU 的处理速度的提升已经大不如前些年,在散热、材料、架构等各方面因素下,CPU 的处理速度已经到达了一个瓶颈。虽然可以通过使用多个 CPU 并行处理数据的方式来解决该问题,但是在资源受限的情况下,没有办法在同一台计算机中放多个 CPU,同时多个 CPU 并行也有较高的技术难度。所以,使用 CPU-GPU 异构,即使用 CPU 去控制 GPU 实现数据处理,便可大大提升计算速度和处理效率,这也是未来高性能计算的发展方向。

1.3 并行图像处理

1.3.1 并行图像处理的应用背景

并行处理技术可以应用到很多领域,其中并行图像处理技术则是并行处理技术应用领域的一个重要分支,而这个领域也是本书讨论的重点所在。

图像处理耗费的时间主要来源于两个方面:一个是算法的复杂程度,算法越复杂处理的时间越长;另一个是需要处理的数据量,其中图像的大小、图像通道数、被处理图像的总数量等都是数据量的体现。

加快图像处理速度,虽然可以通过优化图像处理算法来实现,但是在减小数据量方面是仍然效果有限,处理的数据量较大的算法包括主成分分析[4](Principal Component Analysis,PCA)、独立成分分析[5](Independent Component Analysis,ICA)等。有一些图像处理的算法本身很难在计算简便方面得到优化,如非局部均值(Non-Local Means,NLM)算法[6]。

除了算法复杂度之外,图像本身的像素数量也在急速提升,需要处理的图像尺寸也越来越大。随着近些年电子技术的发展,手机、Pad 等产品都已经可以实现高清摄像功能,图像的清晰度也越来越高,苹果公司在 2016 年 3 月推出的 iPad Pro Mini(iPad Pro 9.7 英寸),摄像头是 1200 万像素,经实测照片是 3024×4032px。日常生活中的图像像素点数量已经开始巨大化,工业科技方面的图像则更是巨大。2012 年,我国第一个高分辨率测绘卫星“资源三号”的日常接收、处理和存储图像信号数据达到了 1790GB[7]。还有近些年异常火热的人工智能、深度学习等领域,同样需要处理大量的图像数据。这样巨大的数据量,让传统串行处理陷入很大的困境,而打破这个困境的有效方法就是采用新的处理方式——并行处理。

1.3.2 并行图像处理的原理

并行计算可以提高计算的效率，虽然不是所有的算法都适合使用 GPU 通过并行处理的技术提高计算速度，但是在图像处理领域，绝大多数的算法都适合使用并行处理技术，可以说并行处理技术在图像处理这一领域有得天独厚的优势。

如图 1-20 所示，假设这是一幅图像的部分像素点，如果现在要将该图像进行处理，将所有的像素点中的数据提高 1 点，那么传统的 CPU 进行处理的方式会按照图 1-21 的方式进行图像处理，将像素点一一列好，然后一个一个地按顺序进行图像处理，如果前边没有处理完成，是不会对后边的像素点进行处理的，这就是传统的串行图像处理。

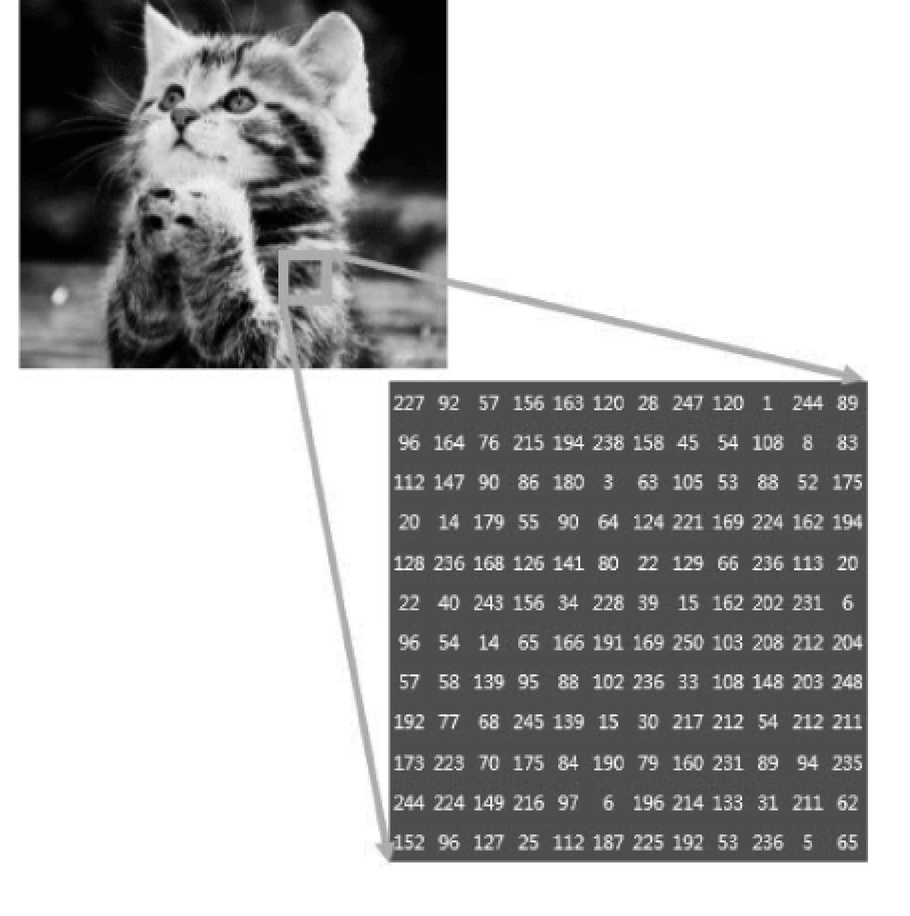


图 1-20　计算机识别图像

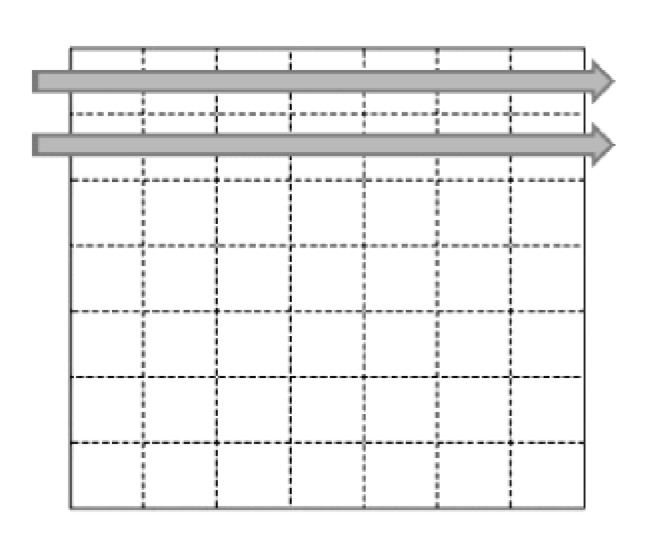


图 1-21　传统的串行处理

并行处理是通过计算机驱动 GPU 多线程并发计算的工作方式，同时对像素点进行计算处理。GPU 内有很多线程，每个线程可以理解为一个单独的个体并且可以进行简单的计算，线程和线程之间是相互独立的，在一些特定的情况下也可以相互通信，这样的工作方式很适合对图像进行处理。图像是由一个个像素点构成的，当时用 GPU 进行图像处理时，可以给每一个像素点分配一个线程进行计算，这个时候 GPU 内的线程并发地处理图像中像素点数据，通过这样的并发式计算可以节省很多处理时间。其计算方式如图 1-22 所示，在一个图像中，每一个像素点对应地分配了一个线程，在分配完线程之后，所有线程同时进行计算。

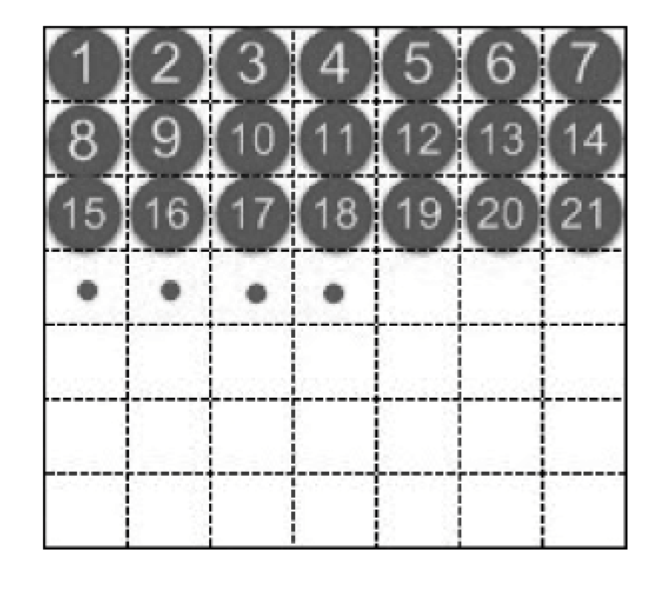


图 1-22　GPU 图像并行计算

对于图像处理，绝大多数的算法都是每个像素点自身或在其他像素点的影响下进行处理，很少会出现一个像素点处理完成之后，其他像素点基于处理后的结果再次进行计算的情况，因此，并行处理技术很适合应用在图像处理这一领域。

1.3.3 并行图像处理的加速效果

本节主要介绍并行图像的处理效果,包括图像处理后的样子、图像处理时间等等。

先给出一个期刊中几位学者在 GPU 图像加速处理领域里边发表的成果。在董荦、葛万成、陈康力三位学者在《信息技术》期刊上发表的数据显示 GPU 在图像处理中的图像锐化、3×3 图像中值滤波算法中有明显的加速[8]。如表 1-1 和表 1-2 所示。

表 1-1 图像锐化

图像分辨率/px	256×256	512×512	768×768	1024×1024	1280×1280	1536×1536
CPU/ms	11.0	43.8	100.0	182.9	273.3	392.1
GPU/ms	26.7	45.3	56.3	108.2	109.4	203.2
数据传输耗时/ms	20.4	29.7	28.1	36.3	35.9	43.7
加速比/ms	0.41	0.97	1.78	1.69	2.50	1.93

表 1-2 图像中值滤波

图像分辨率/px	256×256	512×512	768×768	1024×1024	1280×1280	1536×1536
CPU/ms	198.6	760.8	1679.8	3018.9	4582.8	6734.2
GPU/ms	31.9	59.2	78.2	145.7	173.2	284.3
数据传输耗时/ms	27.1	28.0	26.7	35.0	43.6	48.5
加速比/ms	6.23	12.85	21.48	20.72	26.43	23.69

由表中的数据可知,GPU 的处理速度远远大于 CPU 的处理速度,但是在图像锐化算法处理图像分辨率为 256×256px 时,GPU 的速度会慢于 CPU,相信很多刚刚接触 GPU 开发的读者都会遇到类似的现象:使用 GPU 并行处理数据的速度与单独使用 CPU 处理的速度差不多,甚至比单独 CPU 处理的速度慢。

原因有以下几种:

- CPU 和 GPU 之间的传送数据的时间太长。

GPU 的计算能力是有目共睹的,但是 CPU 和 GPU 之间传送数据的耗时也是非常可观的,如果计算量较小或者是需要 GPU 和 CPU 之间频繁地在传递数据,则应需要注意中间的时间成本,这种传递数据的耗时导致了 GPU 并行处理的总耗时较长。

- 笔记本电脑中的 GPU。

使用笔记本电脑中的 GPU 来进行并行处理时也容易出现 GPU 处理的耗时较长这一情况。台式机的 GPU 有单独的风扇和辅助电路,而笔记本电脑中的 GPU 确实能力有限。如果笔记本电脑与台式机使用同样的 CPU,性能不会有巨大的差异,但是笔记本电脑中的显卡和台式机的显卡不论从散热还是计算能力等方面都会有巨大的差异。如果是专业级别的服务器,这个差别会更大。

- 算法不适合并行计算。

算法不适合使用并行计算,虽然很少,但是确实有一些算法不是很适合用并行计算处理。如果使用并行计算进行加速,固定的时间开销也很高昂,那么使用并行计算效果适得其反。

那么什么样的图像处理算法适合使用 GPU 来进行计算呢?

- 计算量较大。如果处理的时间只有几秒钟,虽然可以用并行处理,但是实际意义不大。

- 图像的数据量较大。图像像素点大、图像的数量多、总体数据量大的算法更适合并行处理。

鉴于以上原因,使用台式机来实现 NLM 算法,通过使用 CPU 和 GPU 的处理时间来进行对比,具体的实现过程将在后面详细介绍。表 1-3 和图 1-23 分别是使用 CPU 和 GPU 来进行 NLM 算法消耗的时间。

表 1-3 CPU 和 GPU 实现 NLM 滤波算法的耗时

图像分辨率/px	64×64	128×128	256×256	512×512	1024×1024
CPU/ms	37.6	109.1	312.7	1251.9	4947.6
GPU/ms	648.2	651.7	654.6	660.1	665.2

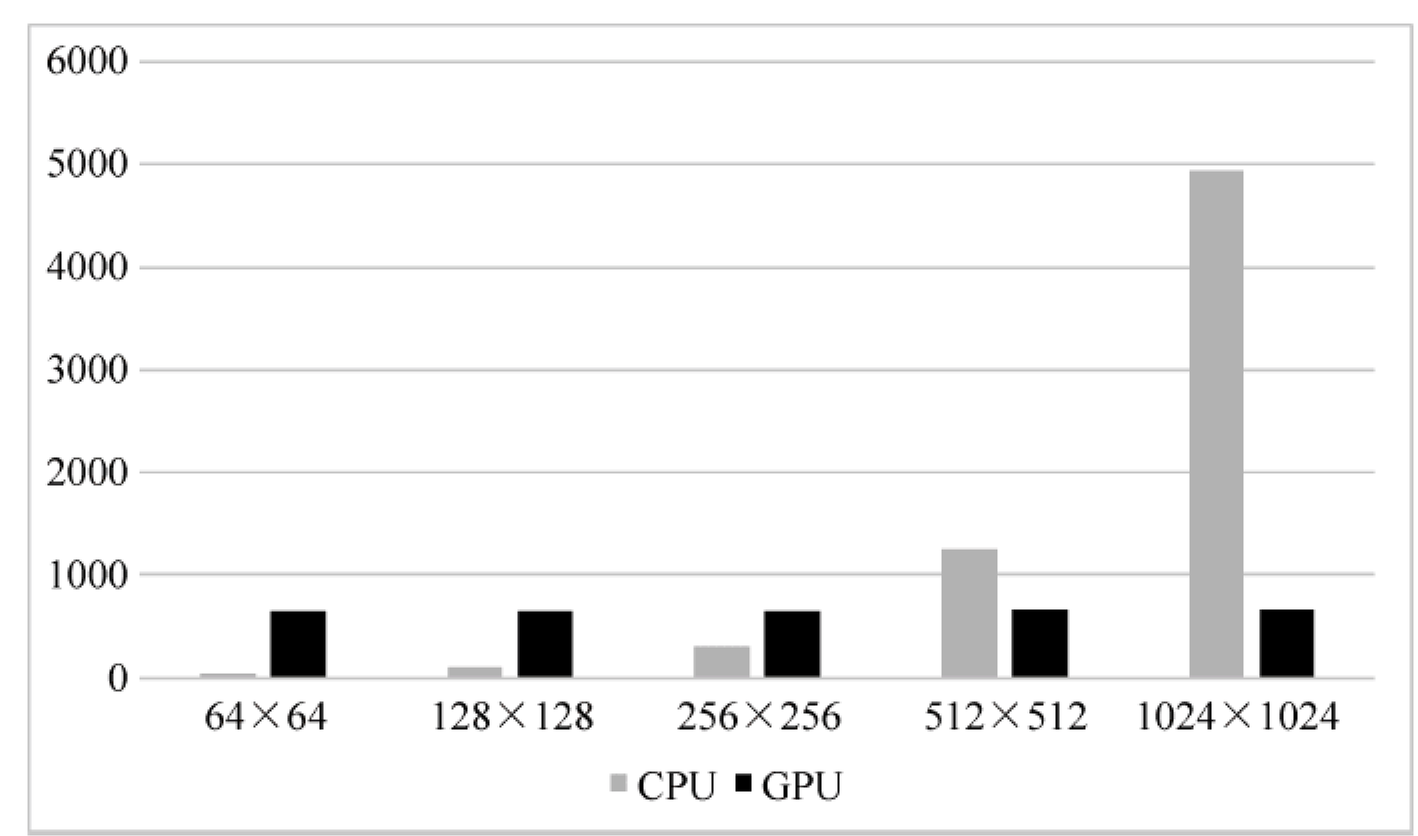


图 1-23 NLM 滤波算法的处理速度

从图 1-23 可以明显看到,随着图像的像素点数量成指数倍增长,CPU 的处理时间也会对应地成指数倍递增。但是对于 GPU,虽然数据量暴增,但是没有明显影响每一次处理消耗的时间。可以说数据量的增多仅仅在 CPU 和 GPU 传递过程中有影响,但是在图像处理的计算能力上,GPU 的处理速度要高于 CPU 的处理速度。

在图像锐化的算法中,CPU 的速度快于 GPU 的速度是因为该算法过于简单,CPU 可以很快实现,GPU 在计算之前需要分配线程,把数据从 CPU 传递给 GPU,计算结束之后需要把数据从 GPU 中返还给 CPU。虽然说 GPU 计算的速度很快,但是一系列数据传递过程还是需要有一定的时间开销,而且这个时间开销是没有办法避免的。而 NLM 算法适合并行处理,其算法本身也有一定的复杂度。当处理的图像像素点数量成指数倍增加时,GPU 仍然只是有分配线程方面时间的开销,几乎不必考虑实际计算过程中所耗的时间,就是说只要在线程够用的情况下,数据量的大小几乎不会影响计算所需要的时间。

1.4 并行图像处理硬件平台

前面已经介绍了并行处理中使用的一些平台软件,在本节将介绍一些硬件方面的基本知识,并给出几种常用硬件型号和对应软件型号。

NVIDIA 公司在市面上常见的 GPU 有三个系列：Geforce 系列、Quadro 系列 Tesla 系列，这三种类型目前均支持 CUDA。近几年 NVIDIA 公司开发的 GPU 都可以完美运行 CUDA，自 2010 年之后出厂的 NVIDIA 的 GPU 都支持 CUDA，现在市面的 NVIDIA 公司的 GPU 都是支持 CUDA 的，以下是早期出品但是仍然支持 CUDA 的 GPU[14]：

GeForce GTX 280 Tesla S1070 Quadro FX 5600

GeForce GTX 260 Tesla C1060 Quadro FX 4700 X2

GeForce 9800 GX2 Tesla C870 Quadro FX 4600

GeForce 9800 GTX+ Tesla D870 Quadro FX 3700

GeForce 9800 GTX Tesla S870 Quadro FX 1700

GeForce 9800 GT Quadro FX 570

GeForce 9600 GSO Quadro FX 370

GeForce 9600 GT Quadro NVS 290

GeForce 9500 GT Quadro FX 3600M

GeForce 8800 Ultra Quadro FX 1600M

GeForce 8800 GTX Quadro FX 570M

GeForce 8800 GTS Quadro FX 360M

GeForce 8800 GT Quadro Plex 1000 Model IV

GeForce 8800 GS Quadro Plex 1000 Model S4

GeForce 8600 GTS

GeForce 8600 GT

GeForce 8500 GT

GeForce 8400 GS

GeForce 8300 mGPU

GeForce 8200 mGPU

GeForce 8100 mGPU

GeForce 9800M GTX Quadro NVS 320M

GeForce 9800M GTS Quadro NVS 140M

GeForce 9800M GT Quadro NVS 135M

GeForce 9700M GTS Quadro NVS 130M

GeForce 9700M GT

GeForce 9650M GS

GeForce 9600M GT

GeForce 9600M GS

GeForce 9500M GS

GeForce 9500M G

GeForce 9300M GS

GeForce 9300M G

GeForce 9200M GS

GeForce 9100M G

GeForce 8800M GTS
GeForce 8700M GT
GeForce 8600M GT
GeForce 8600M GS
GeForce 8400M GT
GeForce 8400M GS
GeForce 8400M G
GeForce 8200M G

1.5 并行图像处理软件平台

本书中使用的并行图像处理依托于 GPU 的并行处理能力，而市面上有很多家公司生产 GPU，主要的开发平台有 NVIDIA 公司推出的 CUDA 和 AMD 公司推出的 OpenCL。CUDA 目前是 GPU 开发最优秀的平台，并且目前很多国内的大公司，例如阿里巴巴、腾讯等，在使用 GPU 集群时都是使用 NVIDIA 公司的 CUDA。因此本书中的 GPU 开发也是采用 NVIDIA 公司推出的 GPU，开发的环境也是该公司推出的 CUDA。

为了实现图像的并行处理，需要在计算机中搭建一个支持并行图像处理的环境。本书中搭建这些环境需要用到如下的五款软件：Visual Studio、TBB、OpenCV、CUDA 和 Cmake，本节将对这五款软件进行简单的介绍以方便后续的学习。

1.5.1 开发平台——Visual Studio

Microsoft Visual Studio（简称 VS）是美国 Microsoft 公司的开发工具包系列产品。VS 是一个基本完整的开发工具集，它包括了整个软件生命周期中所需要的大部分工具，如 UML 工具、代码管控工具、集成开发环境（IDE）等。

所写的目标代码适用于 Microsoft 支持的所有平台，包括 Microsoft Windows、Windows Mobile、Windows CE、. NET Framework、. NET Compact Framework 和 Microsoft Silverlight 及 Windows Phone[9]。

目前最新的版本是 VS 2017，但是目前 CUDA 的最新版本 CUDA 8.0 并不支持 VS 2017，而且 CUDA 7.5 也不支持 VS 2015，所以一般在进行环境搭建时都会选择 VS 2012 或者是 VS 2010，两者的架构分别是. NET Framework 4.0 和. NET Framework 4.5。当二者在计算机中共存时，再使用 VS 2010 就容易出现架构方面的问题，所以建议使用时尽量使用一个版本而不要多个版本一起使用。

1.5.2 计算机视觉库——OpenCV

OpenCV 的全称是 Open Source Computer Vision Library。OpenCV 是一个基于 BSD 许可（开源）发行的跨平台计算机视觉库，可以运行在 Linux、Windows 和 Mac OS 操作系统上。它是轻量级且高效的——由一系列 C 函数和少量 C++类构成，同时提供了 Python、Ruby、MATLAB 等语言的接口，实现了图像处理和计算机视觉方面的很多通用算法。图像处理与计算机视觉是有区别的，图像处理的核心在于对图像中像素点的计算，例如图像分

割、图像融合、图像去噪等等。而计算机视觉是用计算机取代替人的视觉去实现图像特征提取等，最终使计算机模拟出人类的视觉[10]。

使用 C++语言去设计图像处理的算法时，需要图像的读入、像素点的提取等等一系列的工作，都可以通过 OpenCV 这一视觉库完成。目前 OpenCV 最高的版本是 OpenCV 3.2，当 OpenCV 进入到 3.0 系列之后，与之前的 2.0 系列有很大的简化，并且增加了一些新功能。本书为了保证几个软件的兼容性，采用了 2.0 系列比较稳定的版本 OpenCV 2.4.9。

OpenCV 的官网下载地址为 http://opencv.org/downloads.html，2.4.9 版本的具体下载地址如图 1-24 所示。

图 1-24　OpenCV 2.4.9 下载地址

1.5.3　统一设备架构——CUDA

统一计算设备架构（Compute Unified Device Architecture，CUDA）是 NVIDIA（英伟达）公司在 2007 年 6 月提出的一种全新的并行计算架构，可以将数据传入 GPU，并通过 GPU 进行计算的一款软件。支持 CUDA 的硬件设备有 NVIDIA 的 GeForce 系列、ION、Tesla 等，并且 CUDA 现已能在 Windows、Linux 和 Mac 三种系统上完美运行，目前 CUDA 系列的最高版本是 CUDA 8.0。

CUDA 一经推出便得到了极大的欢迎，并且迅速在各个领域中取得了应用。在消费级市场上，几乎每一款重要的消费级视频应用程序都已经使用 CUDA 加速或很快将会利用 CUDA 来加速，其中不乏 Elemental Technologies 公司、MotionDSP 公司以及 LoiLo 公司的产品。在科研领域，CUDA 一直受到热捧。例如，CUDA 现已能够对 AMBER 进行加速。AMBER 是一款分子动力学模拟程序，全世界在学术界与制药企业中有超过 60 000 名研究人员使用该程序来加速新药的探索工作。在金融领域，Numerix 以及 CompatibL 针对一款全新的对手风险应用程序发布了 CUDA 支持并取得了 18 倍速度提升。Numerix 为近 400 家金融机构广泛使用。CUDA 的广泛应用造就了 GPU 计算专用 Tesla GPU 的崛起。全球财富五百强企业现在已经安装了 700 多个 GPU 集群，这些企业涉及各个领域，例如能源领域的斯伦贝谢与雪佛龙以及银行业的法国巴黎银行[11]。

本书在使用 CUDA 时选用的是 CUDA 6.5，不过下载时请一定注意一下，在 CUDA 6.5 及以前的版本会有笔记本（notebook）版本和台式机两种版本，在 CUDA 7.0 以后则没有这种分类了，所以下载时请一定注意这些问题。

CUDA 的下载地址如下：https://developer.nvidia.com/cuda-downloads。

1.5.4　并行编程开发工具——TBB

线程构建模块（Thread Building Blocks，TBB）是 Intel 公司开发的并行编程开发的工具。

Intel 公司在 2004 年开始有了 TBB 的概念，并且在 2005 年成立了专攻 TBB 的团队并且成功在 2006 年的 8 月发布了 TBB 1.0。TBB 作为 Intel 公司众多软件开发工具中的一个，实现了开源供学习和开发，其使用的协议是 GPLv2，本书选用的 TBB 版本是 TBB 4.3[12]。

TBB 的下载地址为 www.threadingbuildingblocks.org/download。

1.5.5 跨平台编译工具——CMake

CMake 软件是一个跨平台的安装(编译)工具,可以用简单的语句来描述所有平台的安装(编译过程)。CMake 这个名字是 Cross Platform Make 的缩写。虽然名字中含有 make,但是 CMake 和 UNIX 上常见的 make 系统是分开的,而且更为高阶。它能够输出各种各样的 makefile 或者 project 文件,能测试编译器所支持的 C++ 特性,类似于 UNIX 下的 automake。只是 CMake 的组态档取名为 CmakeLists.txt。CMake 并不直接建构出最终的软件,而是产生标准的建构档(如 UNIX 的 Makefile 或 Windows Visual C++的 projects/workspaces),然后再依一般的建构方式使用。这使得熟悉某个集成开发环境(IDE)的开发者可以用标准的方式建构其软件,这种可以使用各平台的原生建构系统的能力是 CMake 和 SCons 等其他类似系统的区别之处。

CMake 可以编译源代码、制作程序库、产生适配器(wrapper),还可以用任意的顺序建构执行档。CMake 支持 in-place 建构(二进档和源代码在同一个目录树中)和 out-of-place 建构(二进档在别的目录里),因此可以很容易地从同一个源代码目录树中建构出多个二进档。CMake 也支持静态与动态程序库的建构[13]。

本书在使用 CMake 时,使用了 CMake 的编译源代码的功能,后面构建一个并行的 OpenCV 库,需要使用 CMake,选用的 CMake 型号是 CMake 3.4 版本。

CMake 的下载地址为 https://cmake.org/download/。

1.6 常用软硬件搭配方案

CUDA 正处于高速的发展阶段,很容易出现低版本硬件无法完美运行高版本 CUDA 的情况,也可能会出现高版本硬件的性能无法用低版本的 CUDA 完全开发出来的情况。因此,本节给出几个硬件软件对应的搭配方式,是实验室和自学者比较容易得到的性价比较高的搭配,仅供读者参考,当然如果计算机性能好,还是推荐使用最新版本的 CUDA 等相关软件。

(1) 硬件。CPU:Intel i3-4170M、内存:4GB、GPU:Nvidia GeForce GT 550M。

系统。Windows 7 旗舰版 x64。

软件。VS 2010、OpenCV 2.4.9、TBB 43、CUDA 5.5、CMake 3.4。

(2) 硬件。CPU:Intel i5-3230M、内存:4GB、GPU:Nvidia GeForce GT 750M。

系统。Windows 7 旗舰版 x64。

软件。VS 2010、OpenCV 2.4.9、TBB 43、CUDA 6.5、CMake 3.4。

(3) 硬件。CPU:Intel i7-4710MQ、内存:4GB、GPU:Nvidia GeForce GT 860M。

系统。Windows 10 专业版 x64。

软件。VS 2012、OpenCV 2.4.9、TBB 43、CUDA 7.5、CMake 3.4。

(4) 硬件。CPU:Intel i7-3720MQ、内存:8GB、显卡:GeForceGTX 980 Ti。

系统。Windows 7 旗舰版 x64。

软件。VS 2015、OpenCV 2.4.13、TBB 43、CUDA 8.0、CMake 3.4。

1.7　本书介绍

本书将主要分成四个方面来分别对 OpenCV 和 CUDA 进行介绍。

- 第 1 章主要为相关知识的背景介绍。
- 第 2 章和第 3 章将对机器视觉库 OpenCV 的环境搭建和简单应用进行介绍，采用的处理方式是传统的 CPU 串行处理。
- 第 4 章是 GPU 的结构和 CUDA 环境搭建、简单例程的实现等。
- 第 5 章为如何使用 CUDA 和 OpenCV 来实现图像处理。

1.8　本章小结

本章主要介绍了并行图像处理的基础知识。其中 1.1 节介绍了计算机的硬件构成、计算机中的显卡功能和显卡的发展历史，这些硬件常识对后续的学习和开发有很大的帮助。1.2 节主要介绍了并行计算的概念。1.3 节从并行图像处理的应用背景、并行图像处理的原理以及并行图像处理的加速效果三个角度来介绍并行图像处理的相关知识。1.4 节主要介绍了在计算机上搭建并行图像处理的硬件要求。1.5 节主要介绍了并行图像处理所需的软件：C++开发平台 Visual Studio、计算机视觉库 OpenCV、统一设备架构 CUDA、并行编程开发工具 TBB 和编译工具 Cmake。1.6 节根据前两小节介绍的软硬件，帮助读者合理选择适合的软件版本和硬件型号。

参考文献

[1] https://baike.baidu.com/item/GPU-Z/1583575?fr=aladdin

[2] http://blog.csdn.net/zqm201/article/details/44566535

[3] 赵小川，何灏，吴军. 数字图像处理高级应用：基于 MATLAB 与 CUDA 的实现[M]. 北京：清华大学出版社，2015.

[4] Dong E Z，Fu Y H，Tong J G. Face Recognition by PCA and Improved LBP Fusion Algorithm[J]. Applied Mechanics & Materials，2015，734(12)：562-567.

[5] Draper B A，Baek K，Bartlett M S，et al. Recognizing faces with PCA and ICA[J]. Computer Vision & Image Understanding，2003，91(1 - 2)：115-137.

[6] Wang Y，Huang Z，Yuan Y，et al. Harmonics analysis and simulation of NLM in MMC[C]//Iet International Conference on Ac and Dc Power Transmission. IET，2016.

[7] 李德仁. 我国第一颗民用三线阵立体测图卫星——资源三号测绘卫星[J]. 测绘学报，2012，41(3)：317-322.

[8] 董荦，葛万成，陈康力. CUDA 并行计算的应用研究[J]. 信息技术，2010(4)：11-15.

[9] http://baike.baidu.com/item/Microsoft%20Visual%20Studio?sefr=ps

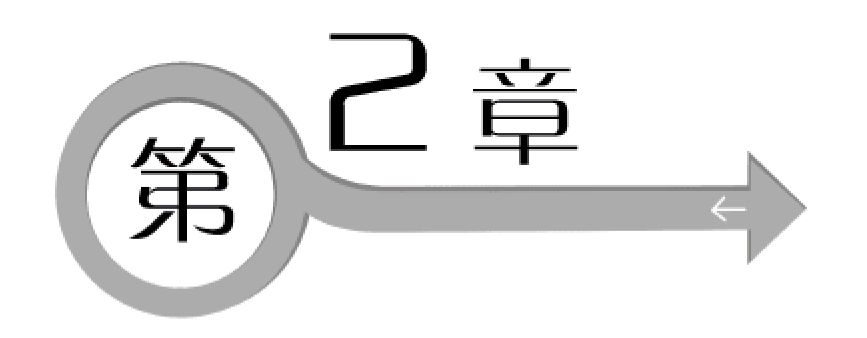
[10] http://opencv.org/

[11] http://baike.baidu.com/item/CUDA?sefr=ps

[12] http://baike.baidu.com/item/TBB?sefr=ps

[13] http://baike.baidu.com/item/cmake?sefr=ps

[14] http://blog.163.com/hlm_87/blog/static/1274798492010893331268/



OpenCV及环境搭建

本章将主要介绍 OpenCV 的发展历程、工作平台和环境搭建，最后补充环境搭建过程中容易遇到的问题。

2.1 OpenCV 的发展历程

OpenCV 是一个计算机视觉库，作为开源代码，OpenCV 自开发以来一直在计算机视觉领域扮演着重要的角色。

OpenCV 最初由 Intel 公司的一个小组进行研发，并于 1999 年 1 月问世，主要的目标是人机界面，能被 UI 调用的实时计算机视觉库，同时可以为 Intel 公司处理器在一些特定的方面做优化。一年后，OpenCV 的第一个开源版本 OpenCV alpha 3 正式发布，之后的几年，OpenCV 迎来了高速的发展，并且开始逐步兼容 Windows、Linux 和 Mac 三大主流系统。在经历过一系列的 OpenCV beta 版本的发展之后，OpenCV 1.0 版本终于在 2006 年 10 月 19 日发布，这个版本就是现在使用的 OpenCV 2 系列和 3 系列的原型。最早期的 OpenCV 1.0 系列并不支持多种语言，但是在 2009 年 10 月推出的 OpenCV 2.0 便开始支持 C++语言作为接口，并且可以在 iOS 和 Android 平台下完美运行，同时也实现了 CUDA 的 GPU 加速，为 Python 和 Java 也提供了语言接口。随着开源代码的一步步优化，终于在 2012 年形成了以稳定和高性能而被全世界公认的 OpenCV 2.4 系列。2014 年 8 月 21 日，OpenCV 3.0 alpha 版本问世；同年 11 月 11 日 OpenCV 3.0 beta 版本问世；在 2015 年 6 月 4 日，发布了 OpenCV 3.0 版本。OpenCV 3 系列使用了全新的内核加插件的架构模式，既保证了自身运行的高稳定性，又优化了原有的代码实现了高效率运行，同时附加的库变得更加灵活多变[1]。

目前 OpenCV 应用的主要领域有人机互动、物体识别、图像分割、人脸识别、动作识别、运动跟踪、机器人、运动分析、机器视觉、结构分析、汽车安全驾驶等领域。

2.2 开发平台——Visual Studio 2010

初步了解 OpenCV 的发展历程后，便需要搭建 OpenCV 所处的开发平台，本书中使用的是 Microsoft 公司推出的 C++编译平台——Visual Studio。

2.2.1 Visual Studio 简介

Microsoft Visual Studio(简称 VS)是美国 Microsoft 公司开发工具包系列产品。VS 是一个基本完整的开发工具集,它包括了整个软件生命周期中所需要的大部分工具,如 UML 工具、代码管控工具、集成开发环境(IDE)等。所写的目标代码适用于 Microsoft 公司支持的所有平台,包括 Microsoft Windows、Windows Mobile、Windows CE、.NET Framework、.NET Compact Framework 和 Microsoft Silverlight 及 Windows Phone。

1997 年,Microsoft 公司发布了 Visual Studio 97,其中包含面向 Windows 开发使用的 Visual Basic 5.0、Visual C++5.0,面向 Java 开发的 Visual J++ 和面向数据库开发的 Visual FoxPro,还包含有创建 DHTML(Dynamic HTML)所需要的 Visual Inter Dev。其中,Visual Basic 和 Visual FoxPro 使用单独的开发环境,其他的开发语言使用统一的开发环境。

1998 年,Microsoft 公司发布了 Visual Studio 6.0。所有开发语言的开发环境版本均升至 6.0。这也是 Visual Basic 最后一次发布,从下一个版本(7.0)开始,Microsoft Basic 进化成了一种新的面向对象的语言——Microsoft Basic.NET。由于 Microsoft 公司对于 Sun 公司 Java 语言扩充导致与 Java 虚拟机不兼容而被 Sun 告上法庭,Microsoft 在后续的 Visual Studio 中不再包括面向 Java 虚拟机的开发环境。

2002 年,随着.NET 口号的提出与 Windows XP/Office XP 的发布,Microsoft 公司发布了 Visual Studio.NET(内部版本号为 7.0)。在这个版本的 Visual Studio 中,Microsoft 剥离了 Visual FoxPro 作为一个单独的开发环境以 Visual FoxPro 7.0 单独销售,同时取消了 Visual Inter Dev。与此同时,Microsoft 引入了建立在.NET 框架上(版本 1.0)的托管代码机制以及一门新的语言 C#(读作 C Sharp)。C# 是一门建立在 C++ 和 Java 基础上的现代语言,是编写.NET 框架的语言。

.NET 的通用语言框架机制(Common Language Runtime,CLR)的功能是在同一个项目中支持不同语言开发的组件。所有 CLR 支持的代码都会被解释成为 CLR 可执行的机器代码然后运行。

.NET 控件是以输入或操作数据为对象。.NET 控件是.NET 平台下对数据和方法的封装,有自己的属性和方法。属性是控件数据的简单访问。方法则是控件的一些简单而可见的功能。过去,开发人员将 C/C++ 与 Microsoft 基础类(MFC)或应用程序快速开发(RAD)环境(如 Microsoft Visual Basic)一起使用来创建这样的应用程序。.NET Framework 将这些现有产品的特点合并到了单个且一致的开发环境中,该环境大大简化了客户端应用程序的开发。包含在.NET Framework 中的 Windows 窗体类旨在用于 GUI 开发,可以轻松创建能够适应多变商业需求的命令窗口、按钮、菜单、工具栏和其他屏幕元素。

Visual Basic、Visual C++ 都被扩展为支持托管代码机制的开发环境,且 Visual Basic.NET 更是从 Visual Basic 脱胎换骨,彻底支持面向对象的编程机制。Visual J++ 也变为 Visual J#。后者仅在语法上与 Java 相同,但是面向的不是 Java 虚拟机,而是.NET Framework。

本书中所有开发的程序都是使用了 Visual Studio 2010 作为开发平台。关于 Visual Studio,目前有众多版本,较旧的有 Visual Studio 2008,还有集大成之作的 Visual Studio

2010 和 Visual Studio 2012，还有比较新的 Visual Studio 2013、Visual Studio 2015 和 Visual Studio 2017。目前业界使用最多，用起来最顺手，并且兼容很多最新软件的平台只有 VS 2010 和 VS 2012[2]。

但这并不是本书选择 VS 2010 的全部原因，VS 2008 版本及其之前的版本对于 OpenCV 的兼容性较差，虽然 OpenCV 的向下兼容的效果很好，但是很多功能无法正常使用。VS 2008 勉强可以算是合格，但再旧的版本就不适用于 OpenCV 了。本书后续需要使用到 CUDA，CUDA 对于 VS 2008 的兼容性也不是很好，所以同时使用 CUDA 和 OpenCV 时，VS 2008 不是很好的选择。VS 2010 和 VS 2012 相似度极高，两者都兼容 OpenCV 和所有版本的 CUDA，因此建议选择这两款开发软件之一作为开发平台。如果仅仅是对 OpenCV 进行开发，可以考虑使用最新版本的 VS 开发工具，但是如果同时使用 CUDA，建议不要使用最新的 Visual Studio 2017。需要注意的是，VS 2013 版本不兼容 Windows 7 系统，如图 2-1 所示。虽然可以通过下载一个插件将 VS 2013 安装在 Windows 7 系统下，但是不建议初学者使用这种方法。VS 2015 不支持 CUDA 7.5，这一点也需要注意。



图 2-1　VS 与 Windows 版本不兼容

2.2.2　安装 Visual Studio 2010

在安装 VS 之前需要确定 VS 版本，建议尽量按照从旧到新的顺序安装。

如果使用的版本较旧，卸载之后换上新版本，新版本会自动升级旧版本的架构，旧版

本不会对新版本造成任何使用上的影响。可是在安装好新版本之后，卸载安装的旧版本VS，有可能会有很多不必要的麻烦出现。如本书中一部分例子，最开始使用了 VS 2012 版本，但最后因为一些原因不能正常使用，卸载后安装了 VS 2010 版本，但是这个过程中同时出现了.NET 架构问题、缺少 MSVCP110D.DLL 问题等，原因是：VS 2010 和 VS 2012 两者的架构分别是.NET Framework 4.0 和.NET Framework 4.5，当计算机中二者共存时再度使用 VS 2010，容易出现架构方面的问题。当新版本被卸载后，新版本的架构将继续存在，这会对旧版本的使用造成较大的麻烦。二者依附的 MSVCP 版本分别是 MSVCP100D.DLL 和 MSVCP110D.DLL，在使用高版本时很容易造成低版本的文件损坏或者覆盖等情况。虽然解决这些问题都不是特别困难，但是处理起来仍然需要很多时间。

最后建议初学者先使用 VS 2010 版本，而且本书搭建的所有环境、例子说明和截图都是使用 VS 2010 版本。VS 2010 的安装过程较简单，以下简略介绍安装步骤：

(1) 双击安装包，进入第一个界面，如图 2-2 所示，单击上方“安装 Microsoft Visual Studio 2010”选项。



图 2-2　VS 2010 安装第一步

(2) 单击之后进入第二个界面加载组件，如图 2-3 所示，建议取消选中“是，向 Microsoft Corporation 发送有关我的安装体验的信息(S)”复选框，单击“下一步”按钮。

(3) 在组件加载完成之后单击“我已阅读并接受许可条款(A)”单选按钮，进行安装即可，如图 2-4 所示。

(4) 之后就进入了安装界面，这个安装过程是将所有的 VS 配件和相关的语言包等安装进去，如图 2-5 所示，首次安装需要接近半个小时的时间，请耐心等候。

(5) 经过等待之后就正式完成 VS 2010 的安装，可以进行下一步的 OpenCV 环境搭建了，如图 2-6 所示为安装之后打开 VS 2010 的界面。



图 2-3　VS 2010 安装第二步

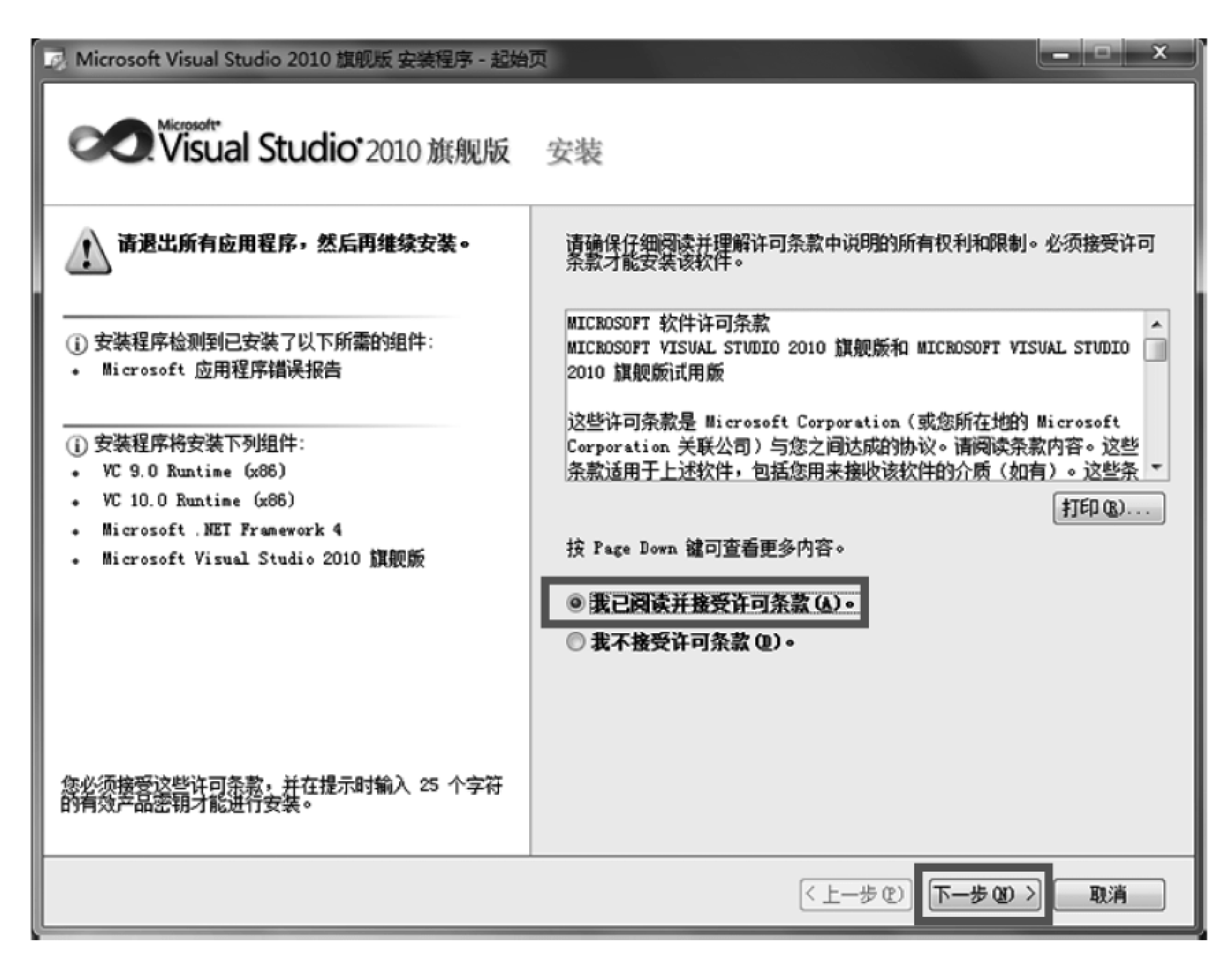


图 2-4　VS 2010 安装第三步



图 2-5　VS 2010 安装第四步



图 2-6　VS 2010 安装成功

2.3　搭建 OpenCV 2.4.9

本节将介绍如何安装 OpenCV，并以 VS 2010 和比较经典的 OpenCV 2.4.9 为例进行环境搭建。OpenCV 自从 1.0 版本开始到现在，搭建的主体思路几乎没有什么变化，因此使用其他版本的 OpenCV 也可以参考此安装流程。

2.3.1 第一步 OpenCV 的下载和安装

成功安装 VS 2010 后即可上网下载需要安装的 OpenCV 2.4.9。OpenCV 建议去官网下载，下载地址和其他常用的网站如下[3]：

OpenCV 下载地址为 http://opencv.org/downloads.html。

OpenCV 官方主页为 https://opencv.org。

OpenCV Github 主页为 https://github.com/opencv/opencv。

OpenCV 开发版 Wiki 主页为 http://code.opencv.org。

下载完成之后，双击 OpenCV 2.4.9 图标选择一个文件夹进行安装。图 2-7 为 OepnCV 2.4.9 的安装包，该图标也是 OpenCV 的 Logo。注意安装路径上不要有中文，所有的文件夹名称都不能用中文。如图 2-8 所示，如果安装的路径中有中文，在安装之后运行过程中会报错。确认好路径之后单击 Extract 按钮进行安装。




图 2-7 OpenCV 2.4.9 安装包

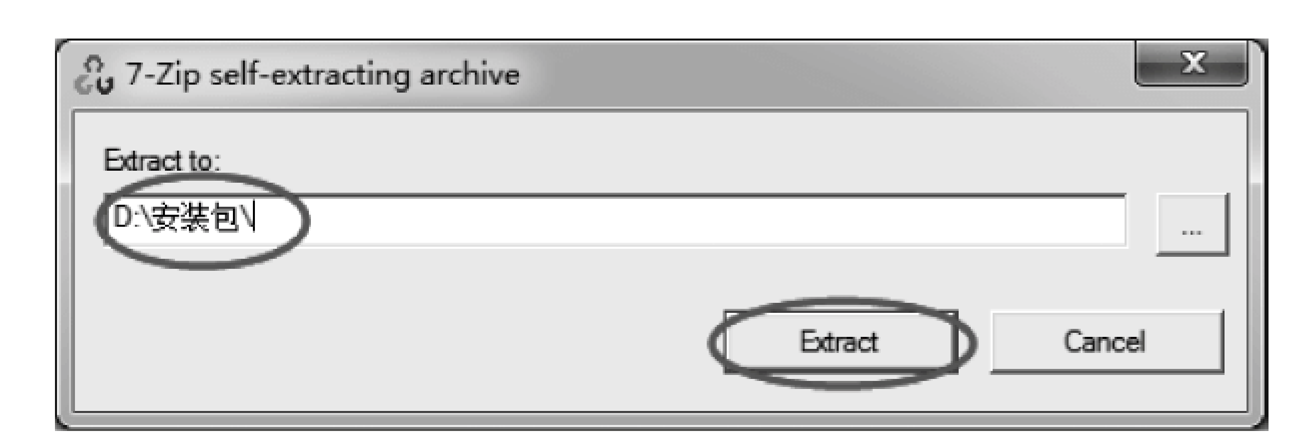


图 2-8 OpenCV 路径中不能有中文

OpenCV 的安装过程其实可以理解成对压缩文件进行解压，因为这个过程没有在系统环境中自动添加路径，安装之后也必须当作一个库，重新添加路径。安装包大约有 300MB，安装之后约 3GB，如图 2-9 所示为安装过程。

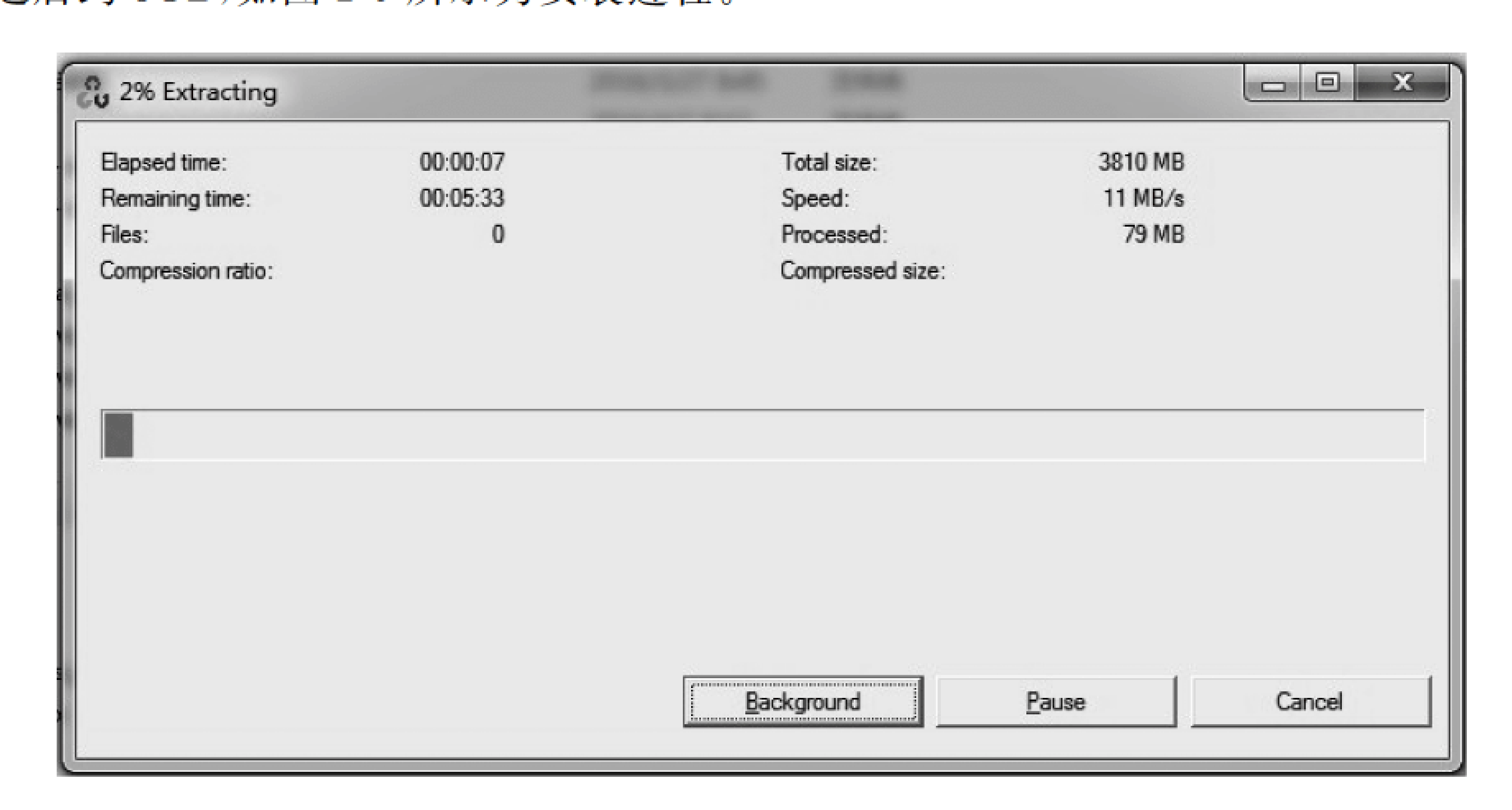


图 2-9 OpenCV 的安装过程

安装之后会有一个 opencv 文件夹，该文件夹内部有 build 和 source 两个文件夹。

build 文件夹中主要为 OpenCV 的 lib 和 dll 库函数，其内部的 x64 和 x86 两个文件夹分别表示编写 64 位程序和 32 位程序需要的库。source 中存放的则是 OpenCV 的例程和一

些其他的相关源代码以及一些说明文档。

2.3.2 第二步 OpenCV 的环境变量配置

(1) 在安装成功 OpenCV 之后，接下来进入 OpenCV 的环境变量配置，右击桌面上的“计算机”图标，通过“属性”命令进入其属性界面，如图 2-10 所示。



图 2-10 配置 OpenCV 环境第一步——计算机属性

(2) 进入属性界面之后在左上部单击如图 2-11 所示的“高级系统设置”选项，进入“环境配置”界面。



图 2-11 配置 OpenCV 环境第二步——高级系统设置

(3) 之后会进入“系统属性”界面，单击“高级”标签，再单击下边的“环境变量”按钮，进入环境变量配置界面，如图 2-12 所示。

(4) 进入“环境变量”界面之后在下边的“系统变量”中寻找 Path 选项,如果没有 Path 选项,则新建一个 Path。之后编辑该选项,在内部把 OpenCV 的路径加进去即可,位置如图 2-13 所示。



图 2-12 配置 OpenCV 环境第三步——进入环境变量配置界面



图 2-13 配置 OpenCV 环境第四步——添加变量值

示例的路径为:

```
D:\OpenCV2.4.9\opencv\build\x64\vc10\bin;
D:\OpenCV2.4.9\opencv\build\x86\vc10\bin;
```

这里的 x86 和 x64 不是说 32 位系统只添加 x86 或 64 位系统只添加 x64,而是指要编译 32 位的程序还是 64 位的程序,示例中计算机的系统是 Windows 7 64 位,所以在使用时会选择将两条路径都添加进入。这样在编程序时就可以在 Win32 和 x64 内自由切换,不会出现任何问题。

在配置路径的过程中,通常需要进行一些文件夹的复制工作。如果想直接复制路径,通常需要顺着路径找到对应的文件夹,例如示例中需要找到 OpenCV 内部的 x86 下的 bin 文件夹。可以先打开需要找到的文件夹,如图 2-14 所示。

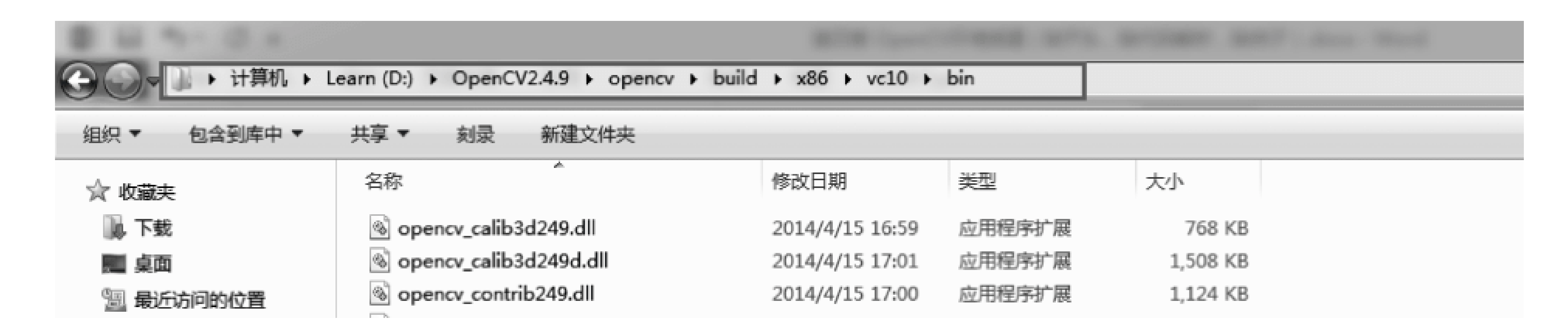


图 2-14 路径复制的小技巧 1

找到文件夹后单击方框上的路径,原本的位置就会变成如图 2-15 所示的样子。

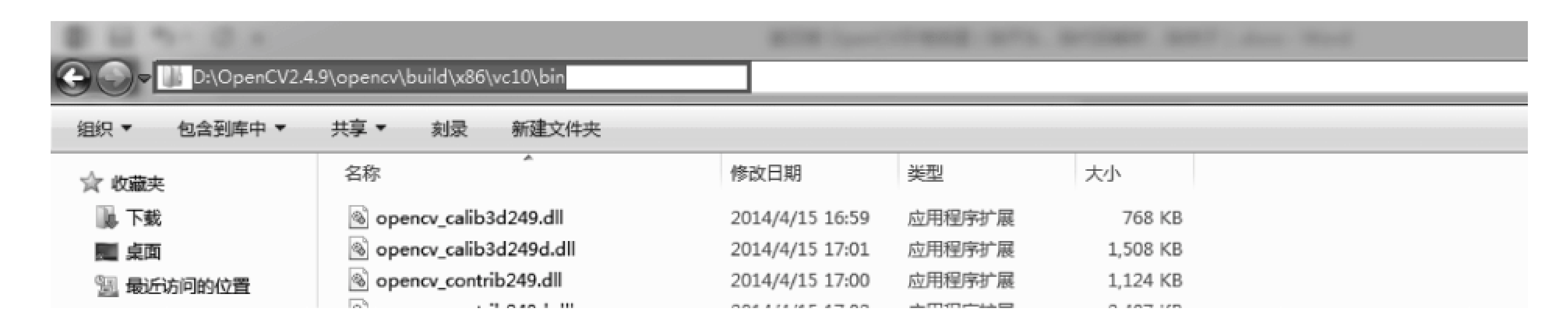


图 2-15 路径复制的小技巧 2

通过这种方式寻找路径就会轻松很多。当然，也可以将复制来的路径放在这个位置，这样能很快地找到对应的文件夹位置。

至此，OpenCV 的添加路径已经完成。但是添加路径时仍有两点需要注意：

• 不要随意改动原本存在的路径。

在添加项目路径之前，不要删掉原本就存在的路径，在最后添加两条全新的即可。所有的路径都必须用“;”隔开，让路径相互之间没有影响。最好养成一种习惯，每次一个路径添加完之后，不论后边还有没有其他路径，都要在结尾加上一个“;”，并且一定是英文输入法中的“;”。不然会导致 OpenCV 无法正常工作，而且在排查的过程中很难找到。

• 选择正确的 vc 版本。

前面示例中的路径为：

```
D:\OpenCV2.4.9\opencv\build\x64\vc10\bin;
D:\OpenCV2.4.9\opencv\build\x86\vc10\bin;
```

在 x86 和 x64 文件夹中都会有 vc10、vc11 和 vc12 这三个文件夹，如图 2-16 所示。

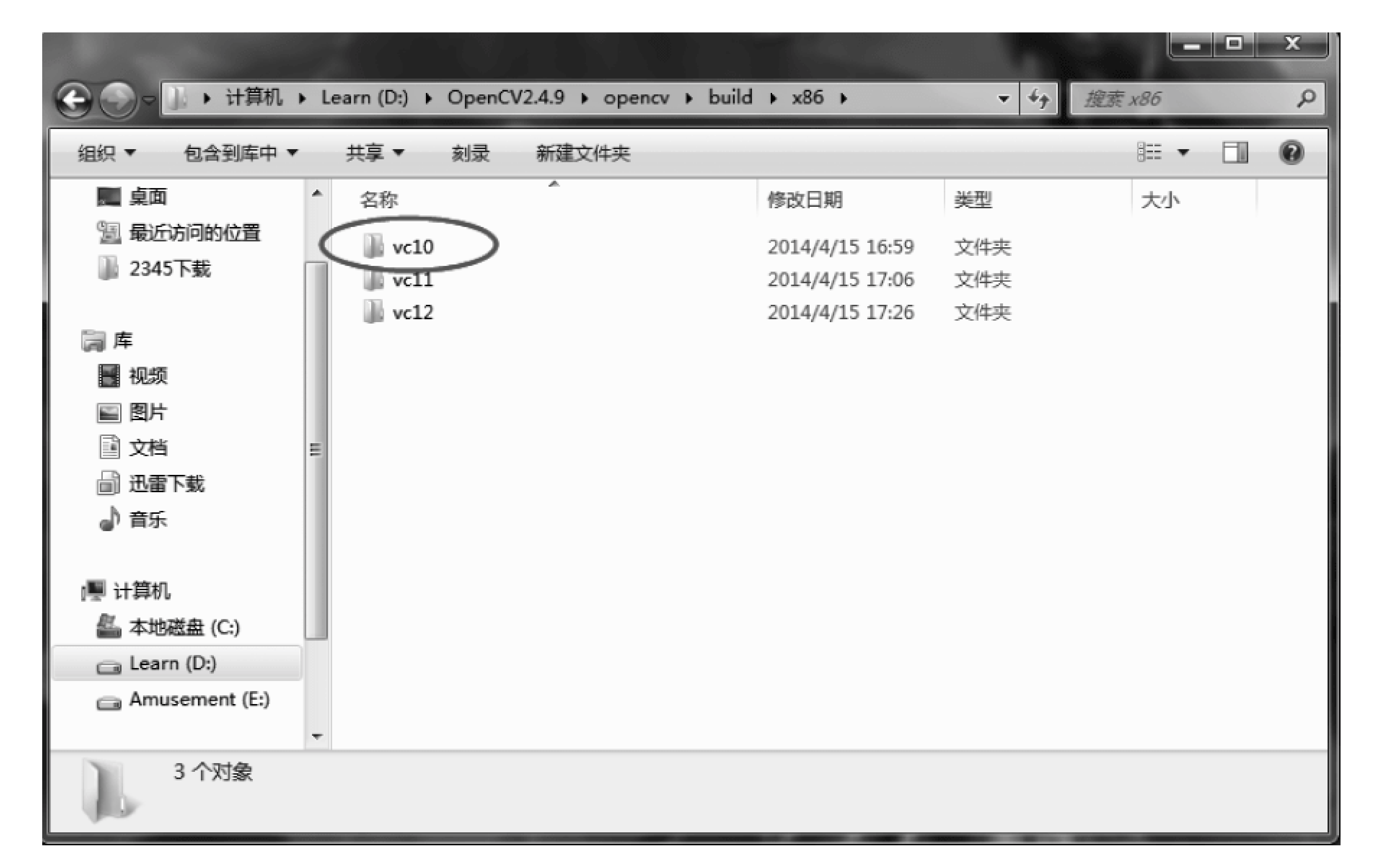


图 2-16 x86 和 x64 内的三个版本文件夹

这三个文件夹是 OpenCV 对应的 VS 版本号，如果使用的是 VS 2010，那么选择 vc10 文件夹的内容。对应关系如下：

vc10 对应 VS 2010；

vc11 对应 VS 2012；

vc12 对应 VS 2013。

如果对应错了，后边的 OpenCV 将无法正常使用。

2.3.3 第三步 工程项目内包含目录的配置

环境变量配置完成后，即可在 VS 2010 内进行编程，本节将详细讲述如何在 VS 2010 内对 OpenCV 库进行配置。

（1）首先打开 VS 2010，新建立一个项目，如图 2-17 所示。虽然使用 Win32 或者 x64 都可以编写程序，打开系统时默认为 Win32 平台，不需要更改什么。为对 x64 进行说明，下面的例子将使用 x64 去实现，单击左上角的“文件”→“新建”→“项目”命令。

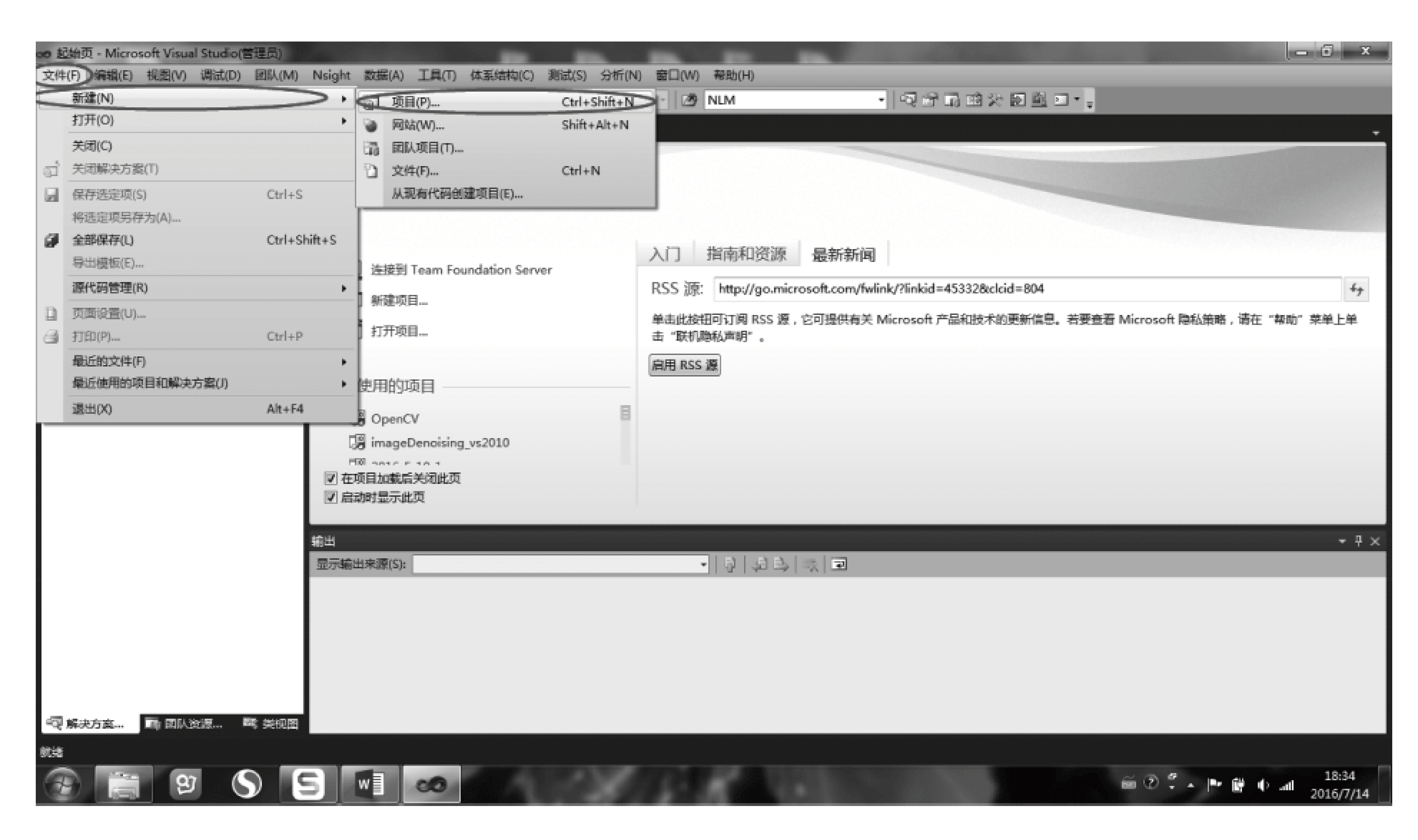


图 2-17 建立 OpenCV 项目文件第一步——创建项目

（2）在模板选项组中选择 Visual C++模板，在该模板中选择“常规”选项，建立一个“空项目”，如图 2-18 所示。下方的名称是这个新建项目的名称和存储位置，建议新建一个文件夹，将练习过程的文件都存在同一个文件夹中，后续学习会更方便。

（3）建立好项目后，在“源文件”上右击，选择“添加”→“新建项”命令。如图 2-19 所示，选择 Visual C++模板，添加一个 C++文件，在下方填上一个 C++文件的名字，该名字支持字母、数字和下画线，但尽量不要出现中文或其他的字符。

（4）回到主界面之后，单击“解决方案平台”菜单栏，如图 2-20 所示，如果其中有 x64 直接选择即可。如果没有这个选项，需要单击“配置管理器”生成一个 x64 平台。如果有，x64 平台直接选中即可跳过步骤（5）；如果没有，需要单击“配置管理器”进入步骤（5）。

（5）单击“平台”选项，选择“新建”选项，如图 2-21 所示。

下拉“新建平台”选择 x64 选项，如图 2-22 所示，单击“确定”按钮即可，然后回到主界面，在 Win32 下拉列表框中找到 x64 平台选项。



图 2-18　建立 OpenCV 项目文件第二步——建立空项目文件



图 2-19　建立 OpenCV 项目文件第三步——选择模板

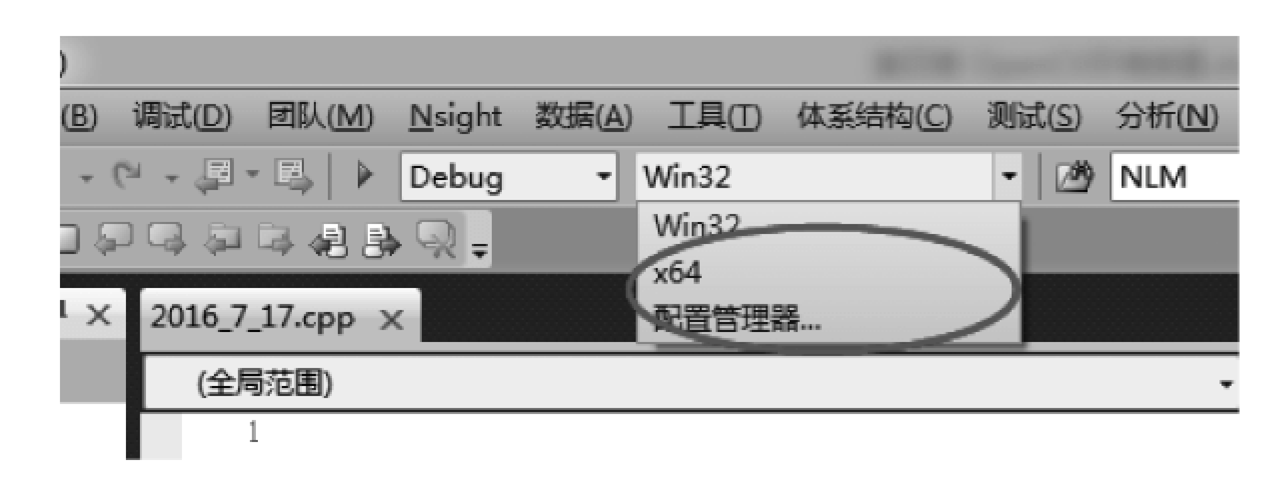


图 2-20　建立 OpenCV 项目文件第四步——配置管理器

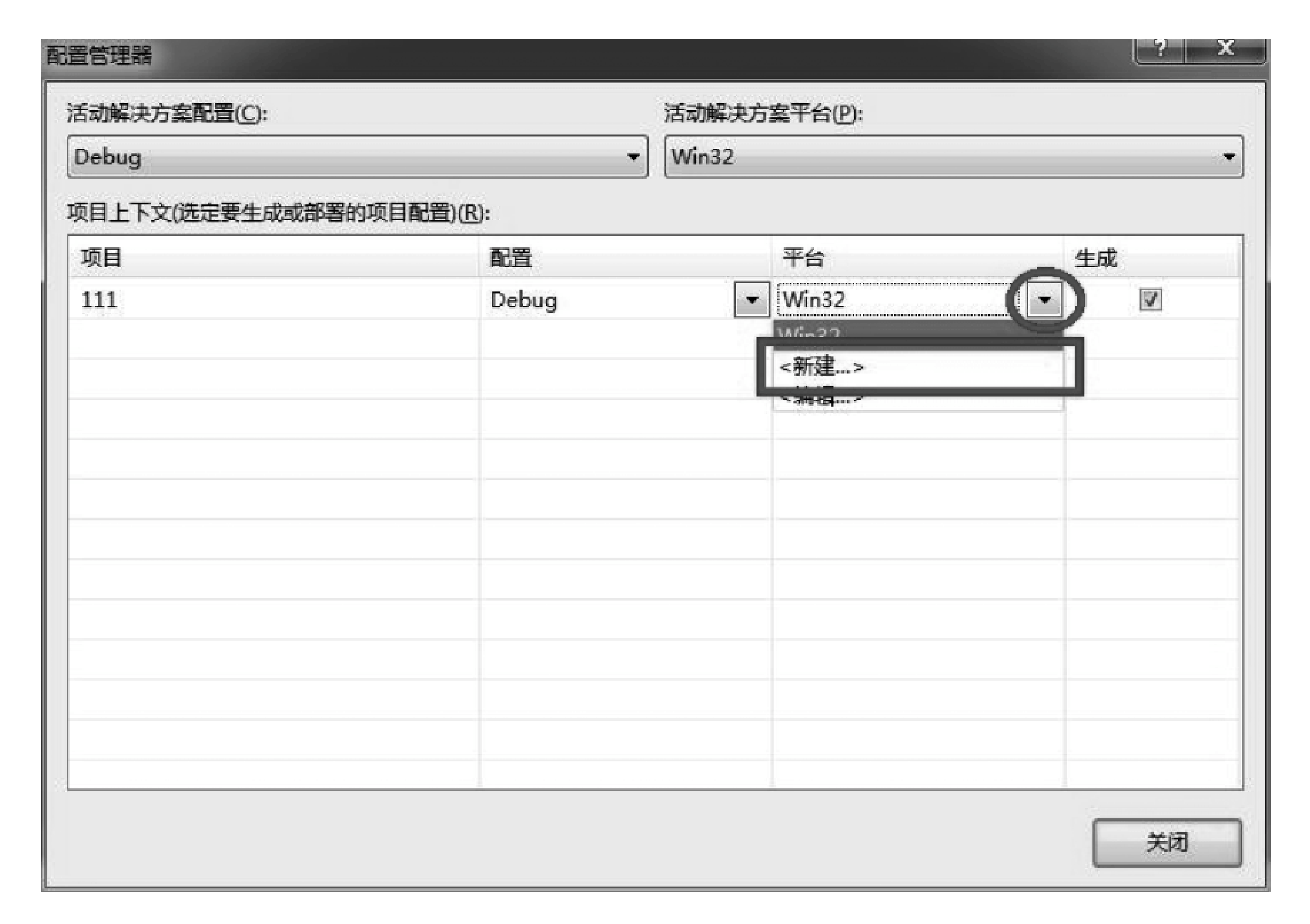


图 2-21　建立 OpenCV 项目文件第五步 1——建立 x64 平台

（6）回到项目文件中，右击项目，选择“属性”命令，在属性界面中进行配置，如图 2-23 所示。



图 2-22　建立 OpenCV 项目文件第五步 2——使用 x64 平台



图 2-23　建立 OpenCV 项目文件第六步——修改项目属性

（7）进入属性页后，在“配置属性”中选择“VC++目录”选项，在右侧的“包含目录”中添加 OpenCV 的 include 路径，这要根据文件存储的位置确定路径，示例的路径有如下三条：

```
D:\OpenCV2.4.9\opencv\build\include
D:\OpenCV2.4.9\opencv\build\include\opencv
D:\OpenCV2.4.9\opencv\build\include\opencv2
```

如图 2-24 和图 2-25 所示。

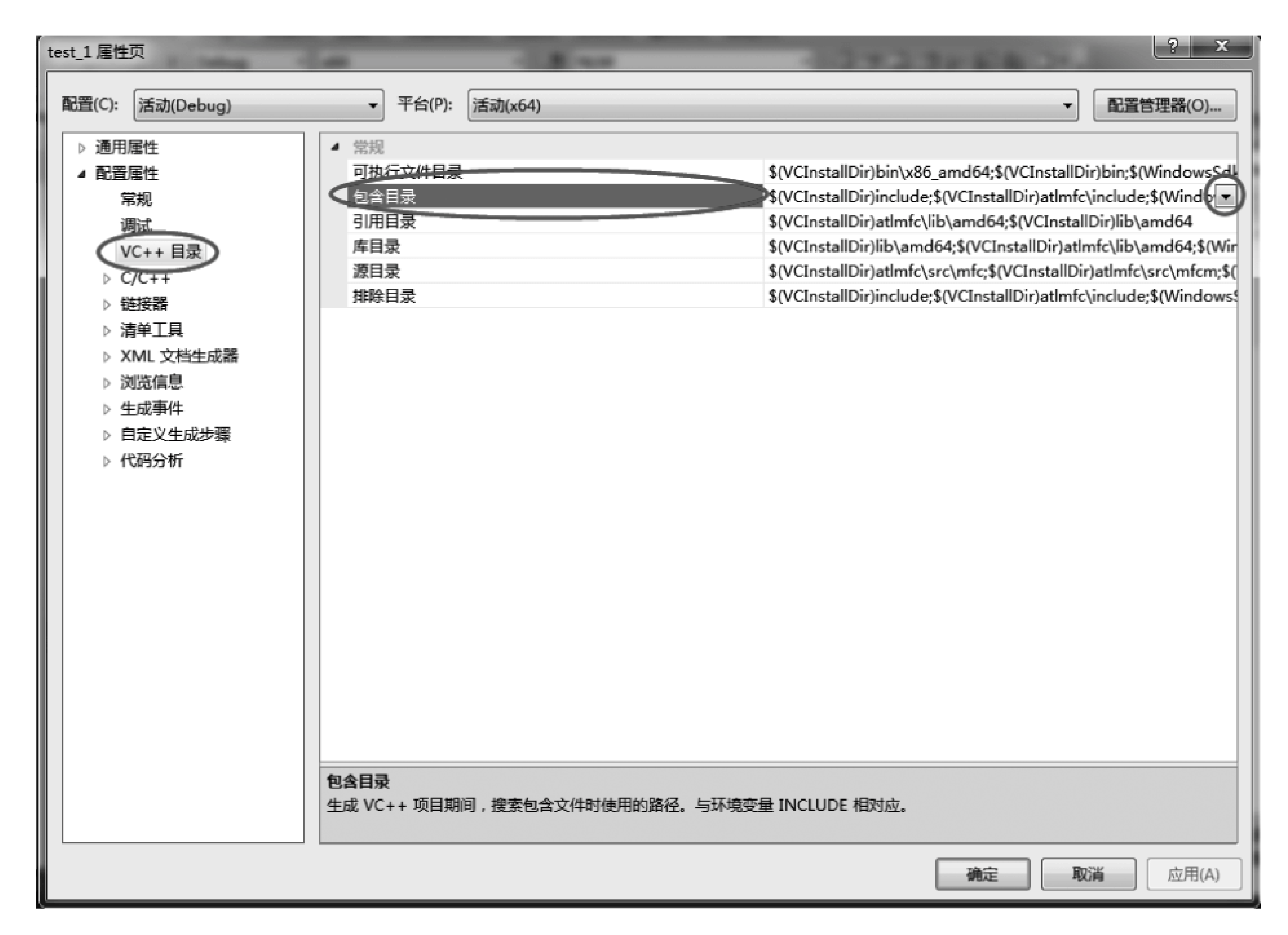


图 2-24　建立 OpenCV 项目文件第七步 1——添加包含目录



图 2-25　建立 OpenCV 项目文件第七步 2——添加包含目录

2.3.4 第四步　库目录的配置

在配置好“包含目录”的路径后进入“库目录”路径配置。“库目录”选项如图 2-26 所示。

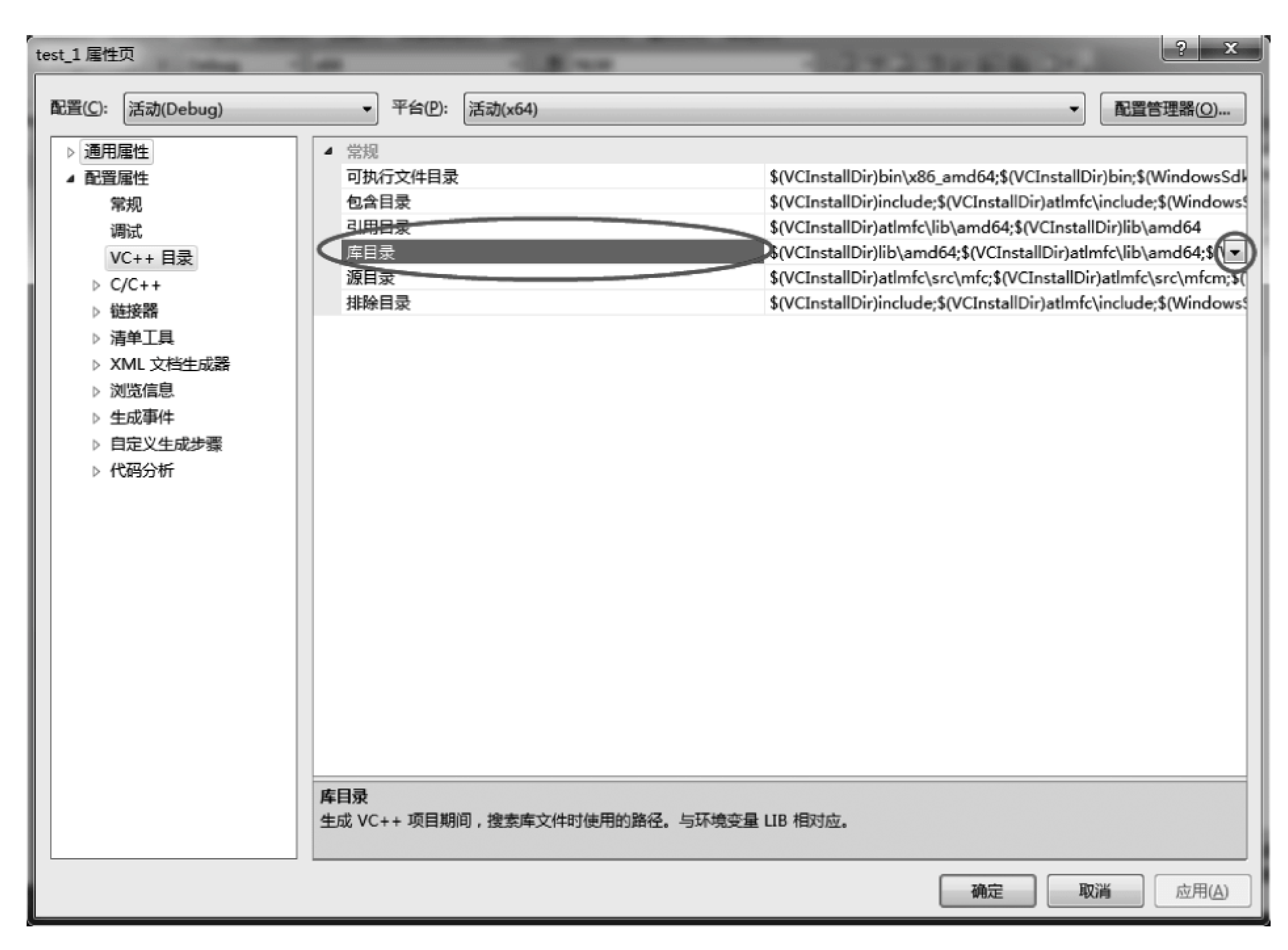


图 2-26　建立 OpenCV 项目文件第八步 1——添加库目录

进入“库目录”编辑界面后，如图 2-27 所示，将 OpenCV 的 lib 路径加入进去，请一定注意 x64 和 x86 的选择，还有 vc10、vc11 和 vc12 的选择。示例的路径为：

```
D:\OpenCV2.4.9\opencv\build\x64\vc10\lib
```

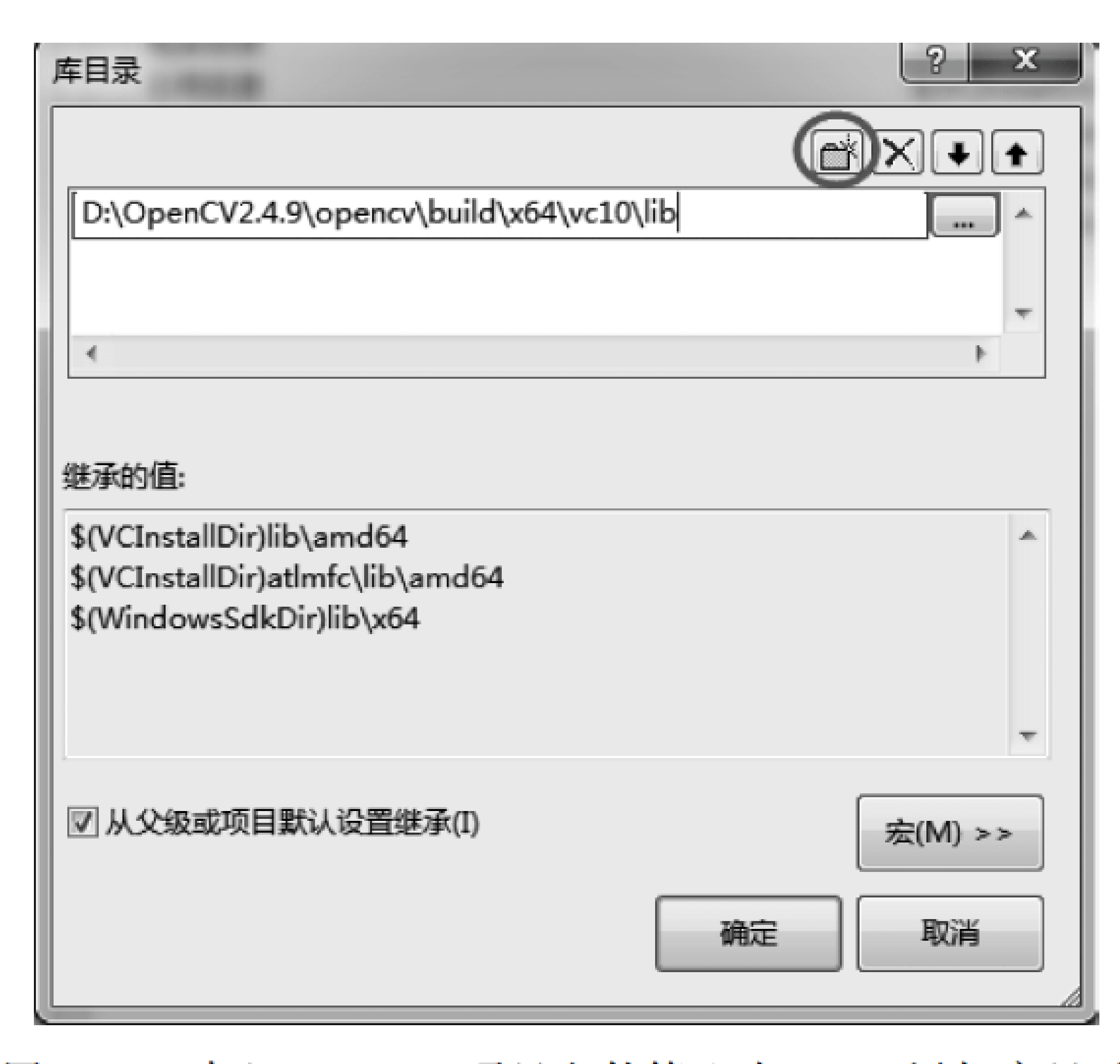


图 2-27　建立 OpenCV 项目文件第八步 2——添加库目录

2.3.5　第五步　附加依赖项的配置

完成“包含目录”和“库目录”的配置后将进行“附加依赖项”的配置。单击“输入”选项，右侧的第一项即为“附加依赖项”，如图 2-28 所示。

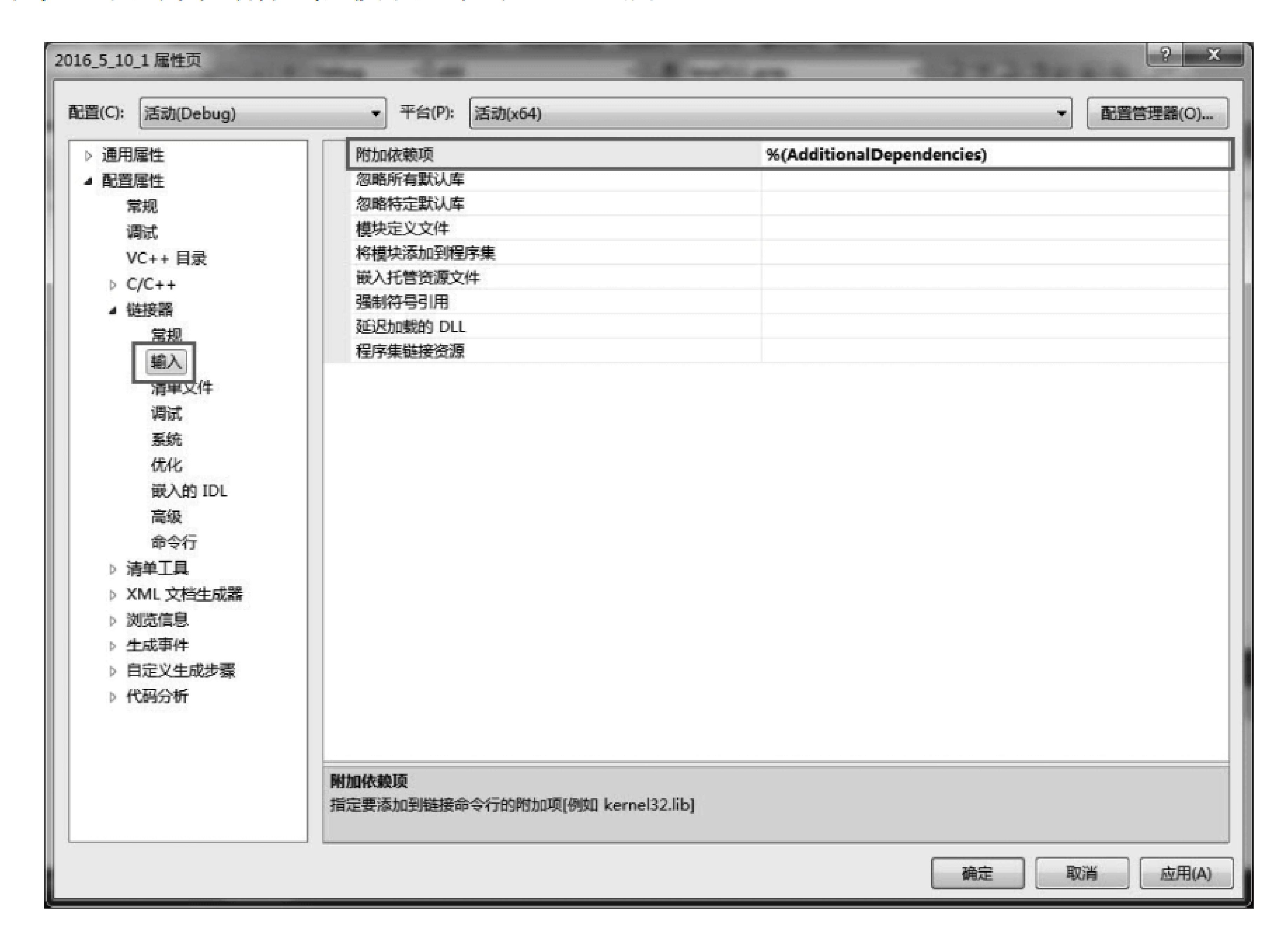
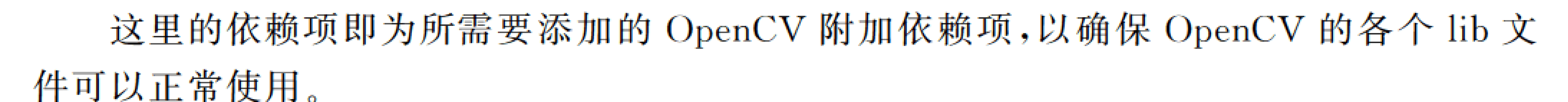


图 2-28　建立 OpenCV 项目文件第九步 1——添加附加依赖项

进入依赖项界面之后要添加 OpenCV 2.4.9 自身的依赖项，如图 2-29 所示，需要添加的依赖项有：

```
opencv_ml249d.lib
opencv_calib3d249d.lib
opencv_contrib249d.lib
opencv_core249d.lib
opencv_features2d249d.lib
opencv_flann249d.lib
opencv_gpu249d.lib
opencv_highgui249d.lib
opencv_imgproc249d.lib
opencv_legacy249d.lib
opencv_objdetect249d.lib
opencv_ts249d.lib
opencv_video249d.lib
opencv_nonfree249d.lib
opencv_ocl249d.lib
opencv_photo249d.lib
opencv_stitching249d.lib
opencv_superres249d.lib
opencv_videostab249d.lib
```



这里的依赖项即为所需要添加的 OpenCV 附加依赖项，以确保 OpenCV 的各个 lib 文件可以正常使用。



图 2-29　建立 OpenCV 项目文件第九步 2——添加附加依赖项

OpenCV 的附加依赖项有其专门的命名方法，通过了解其命名方法可以更好地了解 OpenCV 依赖项，这里以 opencv_ts249d.lib 作为例子进行说明[4]。

opencv_是前缀，紧接着后边的 ts 是依赖项缩写，之后的 249 是版本号，例如 OpenCV 2.4.9 的缩略写就是 249。最后的 d 说明是在 Debug 界面下的依赖项，如果是在 Release 界面下的附加依赖项要将 d 去掉。现在最新的 OpenCV 3.2 版本，则不必添加这么多的依赖项，版本越高优化越好，但是其对于 CUDA 的兼容性还是不如 OpenCV 2.4.9 版本。

假如在搜索附加依赖项的过程中没有搜索到对应 2.4.9 版本的，也可以通过熟悉这个命名的方法解决这个问题，例如现在搜索的是 2.4.8 版本的一个 lib 文件，名称为：

```
opencv_core248d.lib
opencv_features2d248d.lib
opencv_flann248d.lib
opencv_gpu248d.lib
```

如前所述，可以将结尾的几个数字改正，即将程序改成对应 2.4.9 版本的 lib 文件，如下所示：

```
opencv_core249d.lib
opencv_features2d249d.lib
opencv_flann249d.lib
opencv_gpu249d.lib
```

2.3.6　第六步　清单项配置

添加完附加依赖项后，建议将“嵌入清单”选项关闭，当然，这一项如果不关闭也不会影响程序运行的结果，但是在处理过程中会出现很多警告(warning)，影响处理时间，所以还是建议关闭“嵌入清单”选项。

单击“清单工具”选项，在右边寻找“嵌入清单”，将原本的“是”改成“否”即可，如图 2-30 所示。

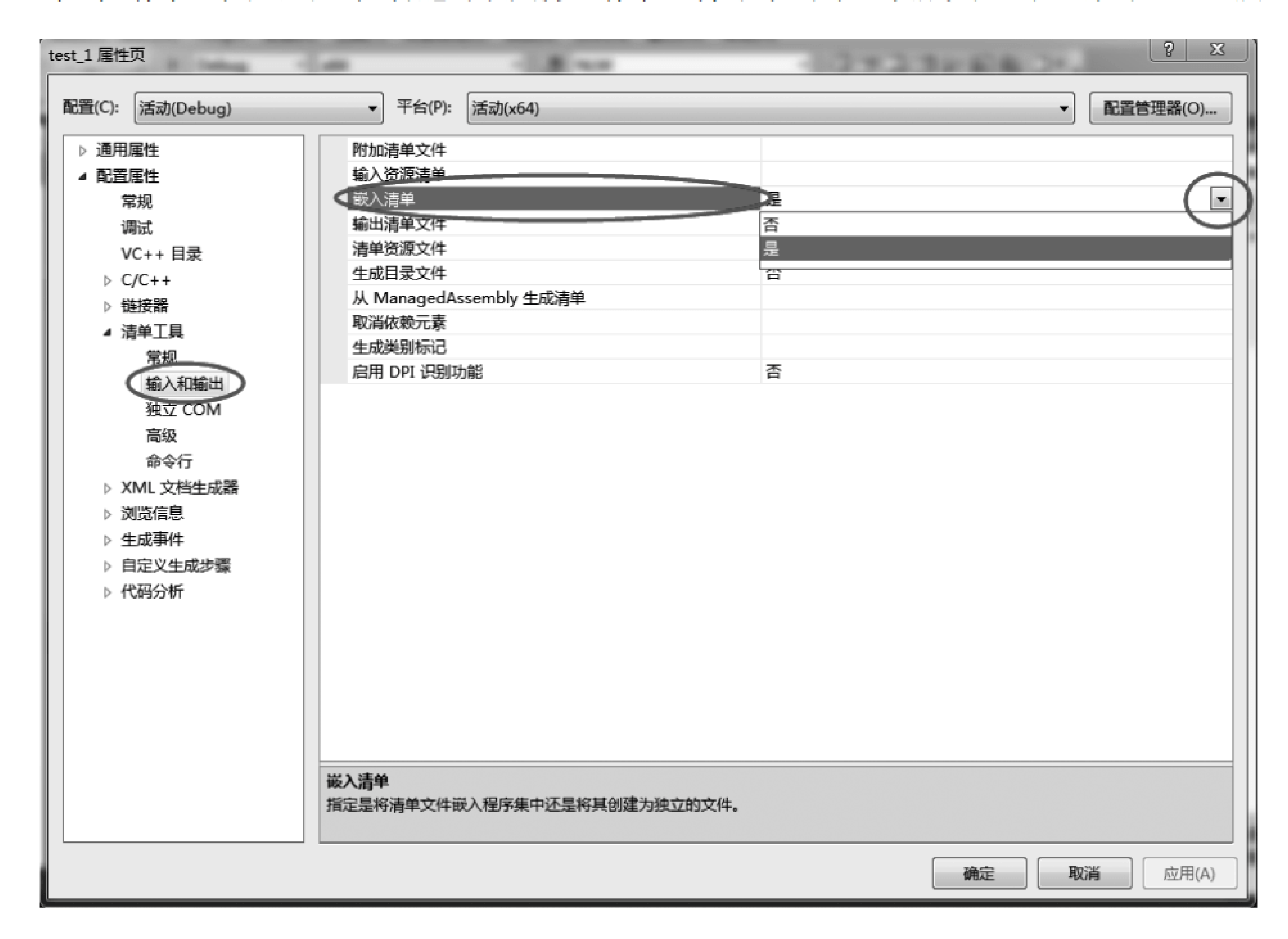


图 2-30　建立 OpenCV 项目文件第十步——修改嵌入清单

2.3.7　第七步　Release 配置

至此，Debug 配置全部完成，在不配置 Release 的情况下不会影响 Debug 的正常调试。不过为了后续的学习，仍然需要掌握 Release 的配置。如图 2-31 所示，在“属性页”左上角的“配置”下拉列表框中将 Debug 选项改成 Release。

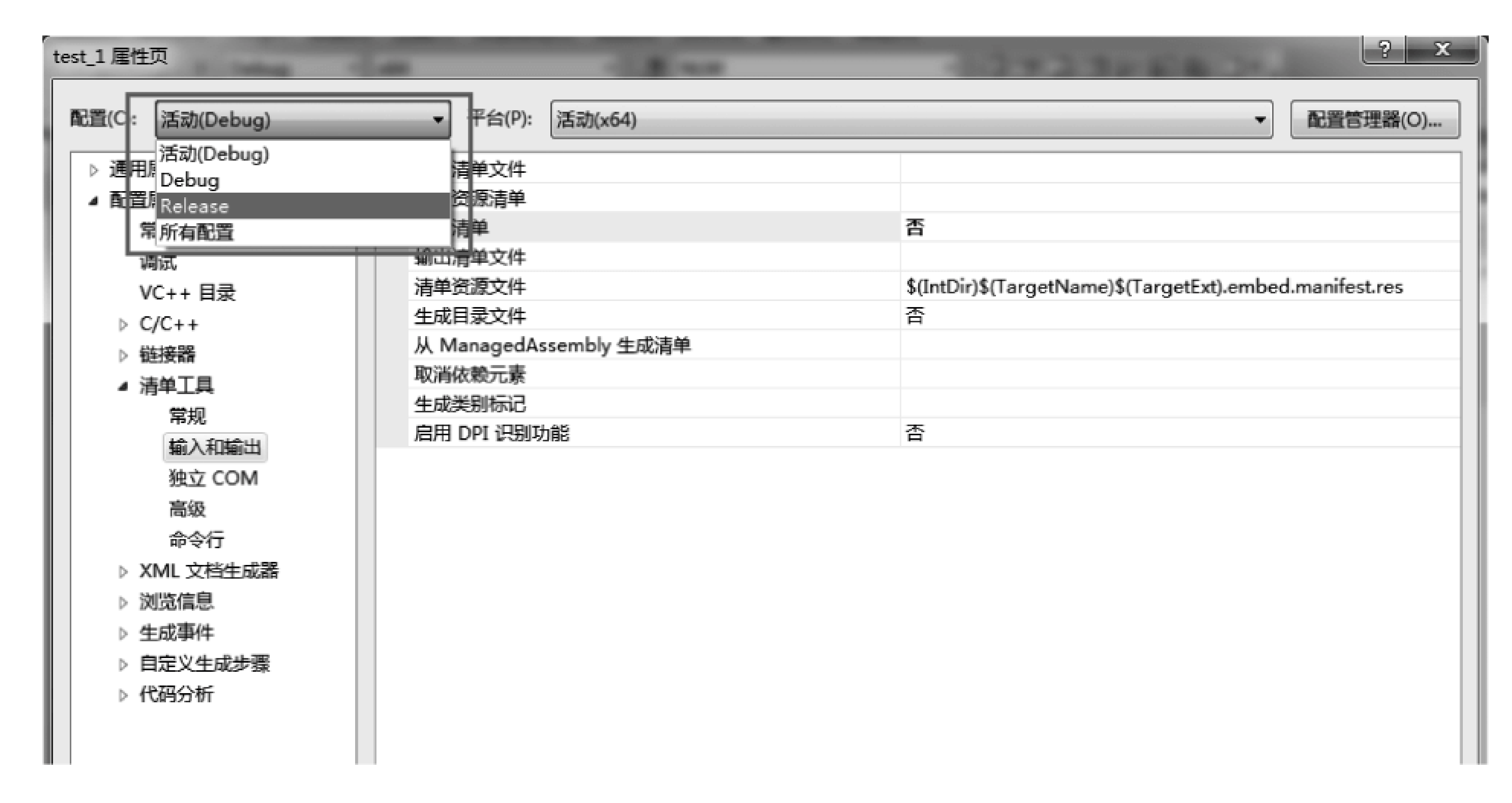


图 2-31　建立 OpenCV 项目文件第十一步 1——Release 属性配置

这时需要在 Release 界面内重新进行配置，进入 Release 界面后重新走一遍第三步、第四步、第五步和第六步，但是第五步要稍稍有些变化。

第五步中的附加依赖项需要将 lib 文件由 Debug lib 改为 Release lib，即将添加的依赖项的 d 去掉，如图 2-32 所示，即变成：

```
opencv_objdetect249.lib
opencv_ts249.lib
opencv_video249.lib
opencv_nonfree249.lib
opencv_ocl249.lib
opencv_photo249.lib
opencv_stitching249.lib
opencv_superres249.lib
opencv_videostab249.lib
opencv_calib3d249.lib
opencv_contrib249.lib
opencv_core249.lib
opencv_features2d249.lib
opencv_flann249.lib
opencv_gpu249.lib
opencv_highgui249.lib
opencv_imgproc249.lib
```

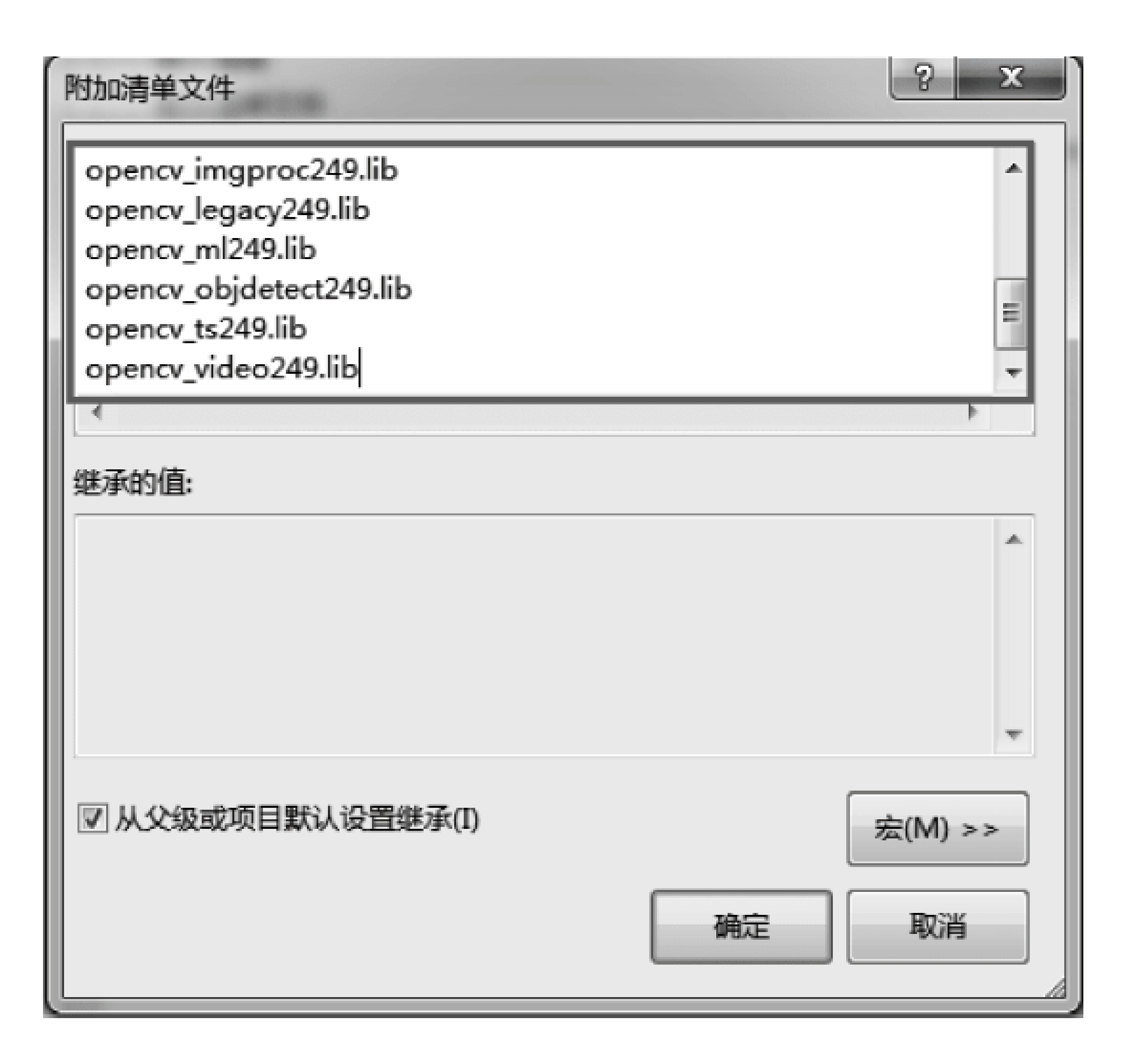


图 2-32　建立 OpenCV 项目文件第十一步 2——Release 内附加依赖项

2.3.8　第八步　加入 OpenCV 动态链接库

在完成上述的配置后，还需要将对应的 lib 文件都复制到相应的 Windows 文件夹中，否则很容易出现如图 2-33 所示的文件缺失型错误“丢失 opencv_core249d.dll”。手动依次添加每一个 lib 文件会更加麻烦，所以直接将所有的相关 lib 文件全部复制到对应的 Windows 文件夹中即可避免这个问题。

图 2-33　建立 OpenCV 项目文件第十二步 1——dll 文件缺失

首先找到 OpenCV 安装的位置，找到 build 文件夹，其中会有 x86 和 x64 两个文件夹，再将 x86 文件夹打开，找到对应 VS 版本的文件夹，如示例中使用的是 VS 2010，那么就选择 vc10 文件夹，之后单击 bin 文件夹，寻找对应的 dll 文件，如图 2-34 所示。

图 2-34　建立 OpenCV 项目文件第十二步 2——dll 文件复制

把这个文件夹中所有的内容全选并复制，粘贴到 C 盘中 Windows 下的 System32 文件夹中即可解决这个问题。

示例中的 OpenCV x86 的 bin 路径为：

```
D:\OpenCV2.4.9\opencv\build\x86\vc10\bin
```

相对地，将要复制到的 Windows 文件夹路径为：

```
C:\Windows\System32
```

在完成上述的 x86 lib 文件的复制工作后，回到 OpenCV 的文件夹中，找到和 x86 相对的 x64 文件夹，然后在 x64 文件夹中找到 vc10 文件夹，进入之后将 bin 文件夹中所有的文件进行复制，并粘贴到 C 盘中 Windows 下的 SysWOW64 文件夹中。将 VS 软件关闭，重新启动 VS 软件，即可完成在 Windows 文件夹下的 OpenCV 动态链接库的配置。

示例中的 OpenCV x64 的 bin 路径为：

```
D:\OpenCV2.4.9\opencv\build\x64\vc10\bin
```

相对地，应该粘贴的地址为：

```
C:\Windows\SysWOW64
```

2.3.9 第九步 环境测试

完成上述所有的配置后，进入最后的环境测试步骤。回到最初的VS界面，找到“源文件”下已经配置好的.cpp文件，之后在右侧编写程序即可，如图2-35所示。

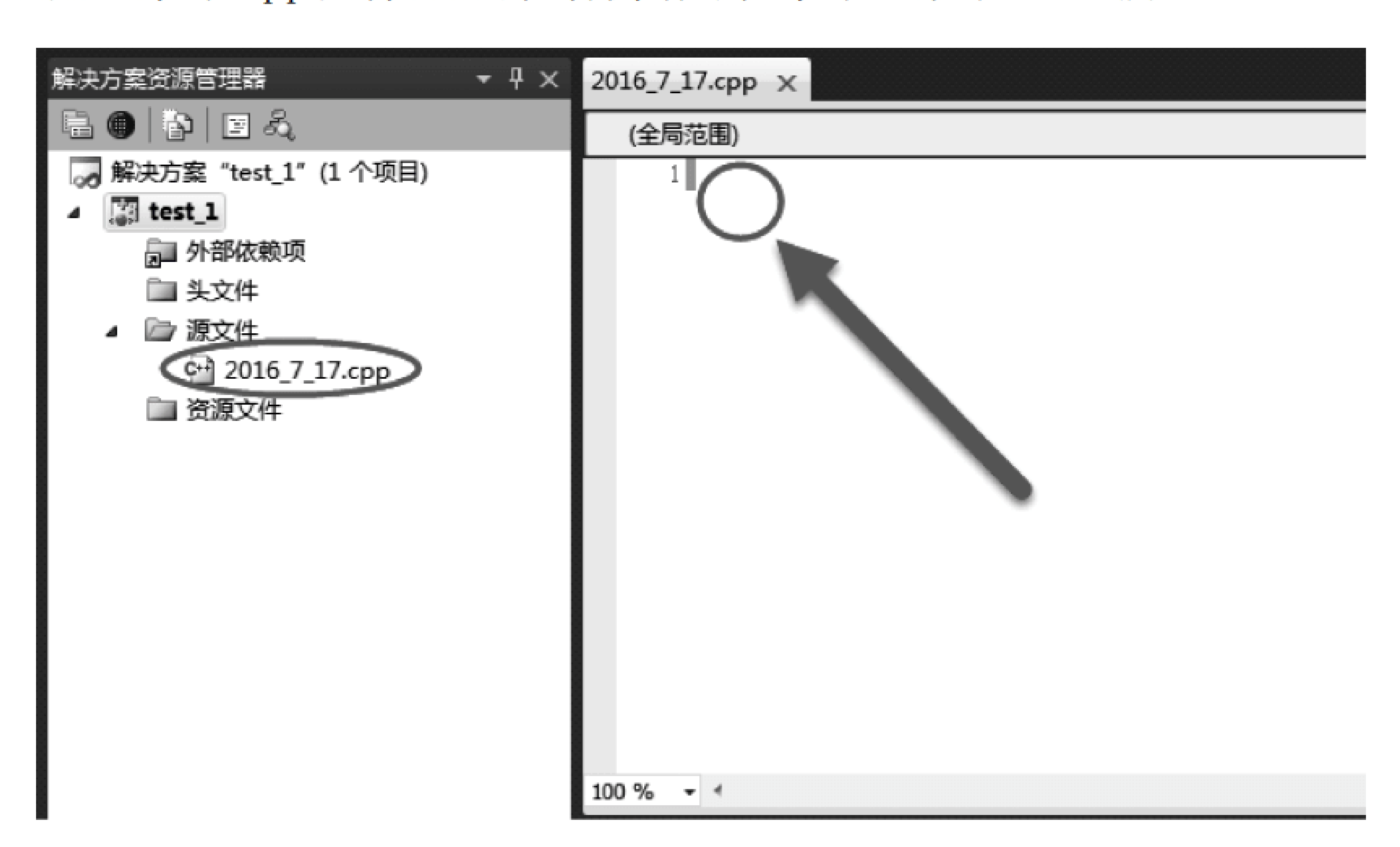


图2-35 建立OpenCV项目文件第十三步1——环境测试

给出一段比较简单的OpenCV图像显示例子，作为环境测试，程序实现如下：

```
#include <opencv2/opencv.hpp>
using namespace cv;

void main()
{
    Mat srcImage = imread("Peashooter.jpg");
    imshow("图像显示",srcImage);
    waitKey(0);
}
```

这段程序是将一个图像进行读入然后再显示出来，如果这段程序正确地显示出来，则证明OpenCV已经完全配置成功并且可以正常使用。

实现结果如图2-36所示。

使用imread函数读入图像有两种写法：一种是将图像放入默认路径中，直接输入图像的名称进行读入；另外一种是直接给出图像存储的路径，后面会对imread函数进行较详细的介绍。

第一种方式就是示例中在默认路径下读入图像，如下：

```
Mat srcImage = imread("Peashooter.jpg");
```

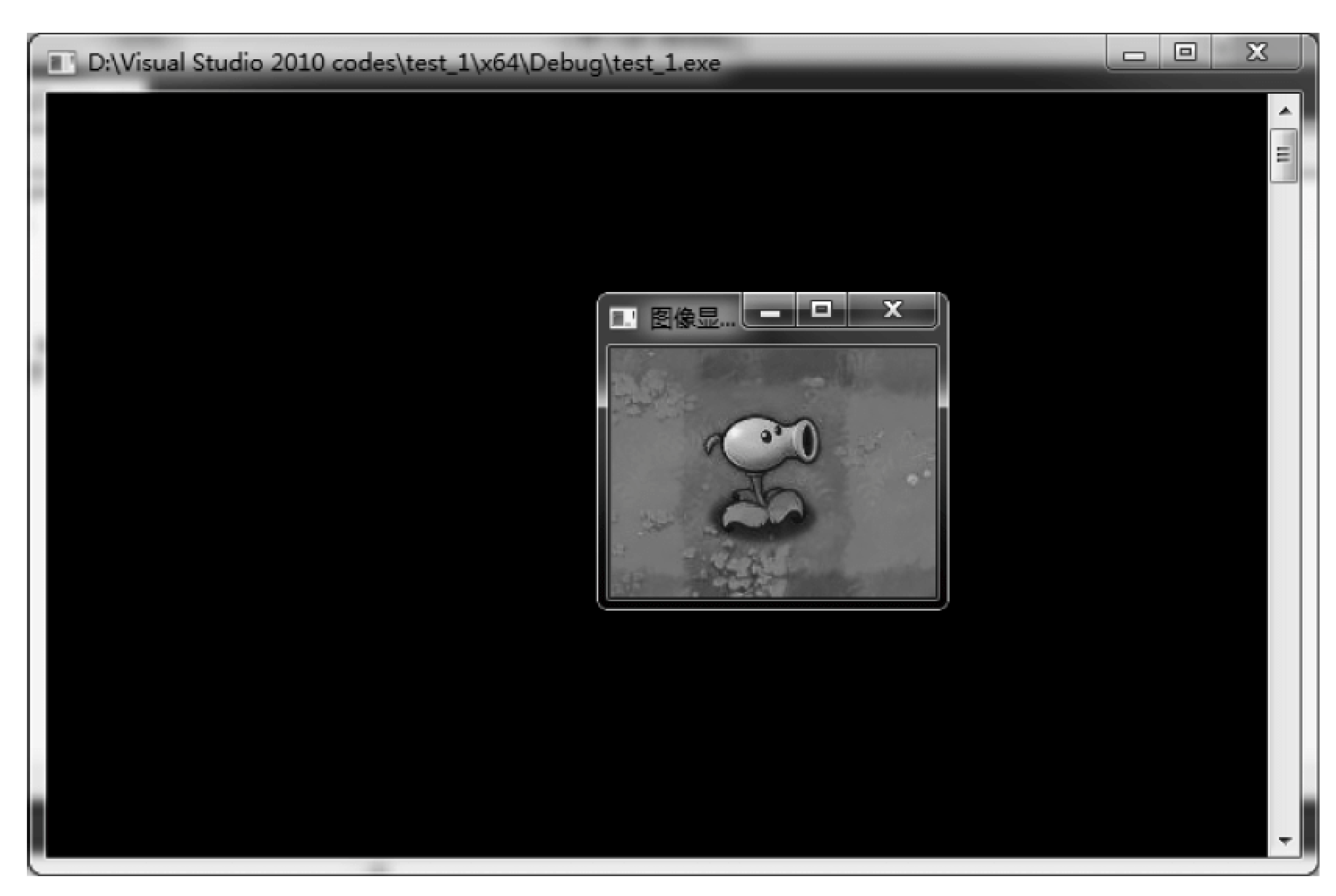


图 2-36　建立 OpenCV 项目文件第十三步 2——环境测试结果

使用这种方式读入图像要求图像必须放在默认路径下，以下提供一个小技巧可以快速找到默认路径，如图 2-37 所示。右击读入图像的.cpp 文件，如示例中的 2016_7_17.cpp，之后单击"打开所在的文件夹(O)"命令，就可以直接进入该文件默认的路径了。

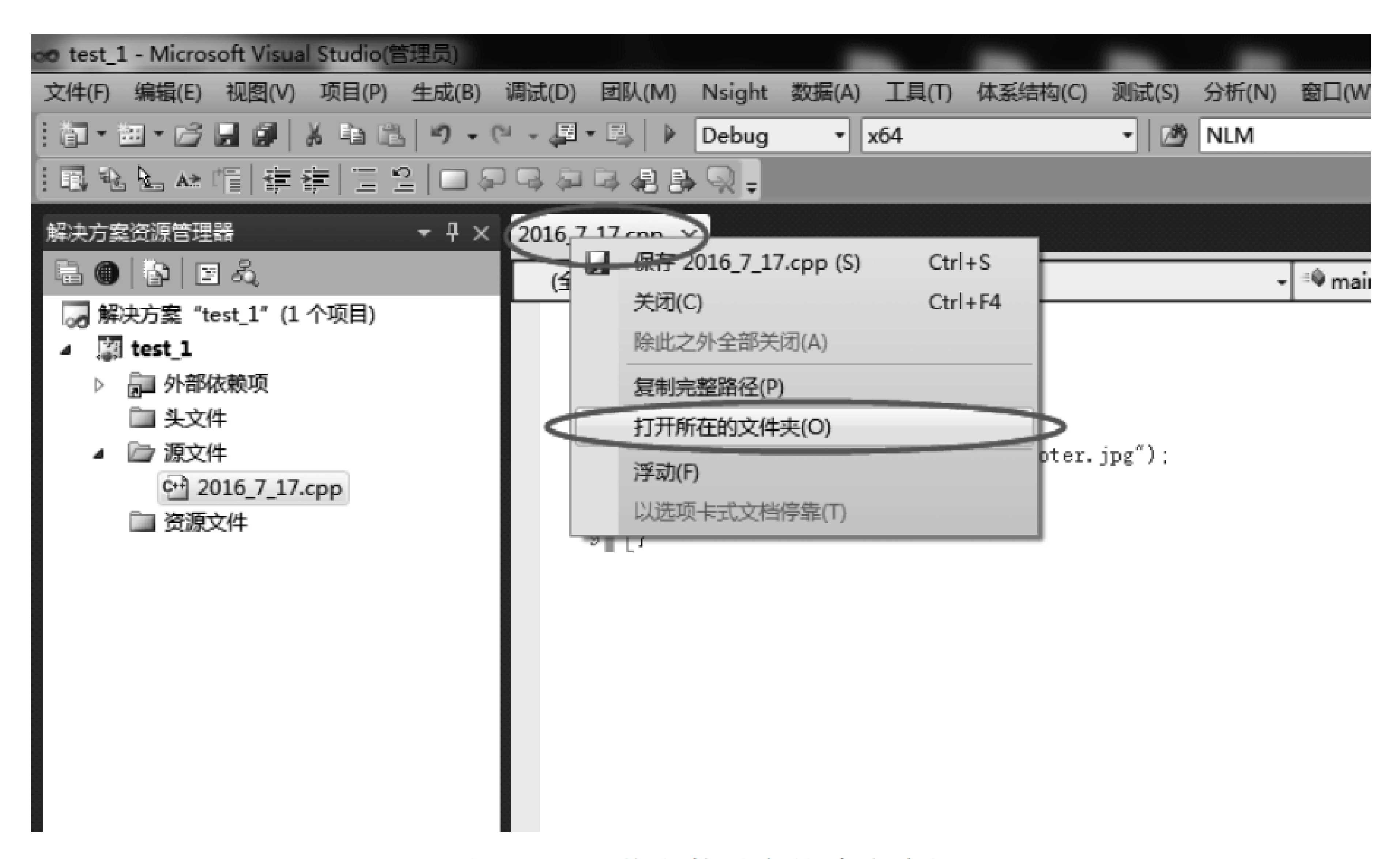


图 2-37　寻找文件对应的默认路径

进入该文件夹之后，将所需要的图片文件复制进去即可，如图 2-38 所示，箭头指向的图标即是所需要的图像。

第二种方式就是直接将文件存储的路径写出来，如下：

```
Mat srcImage = imread("D:\Peashooter.jpg");
```



图 2-38　将图像存放到默认文件夹

这种写法就是在指定的路径下寻找图片，上述写法就是在 D 盘中找 Peashooter.jpg 文件，当然也可以加深文件夹，如下：

```
Mat srcImage = imread("D:\zlase\pee\Peashooter.jpg");
```

这种读取的方式也可以成功实现图像的读入，但仍然要求存储图像的路径中不能包含带中文名称的文件夹。

建议使用 imread 读入图像时采用第一种方式，第二种方式有可能因为文件夹过深而没办法成功读取图像，并且第二种读入方式相比第一种更耗时。

2.4　OpenCV 基本架构

在编程开发之前，最好先了解一下 OpenCV 的基础架构，这样更有利于编程和开发。

在已安装的 OpenCV 文件夹中按路径找到 include 文件夹，并找到其中的 opencv 和 opencv2 文件夹。opencv 文件夹中包含的是最基础也是最常用的 OpenCV 1 系列的头文件，如图 2-39 所示。

cv.h	2013/12/20 17:49	C/C++ Header	4 KB
cv.hpp	2013/12/20 17:49	C/C++ Header	3 KB
cvaux.h	2013/12/20 17:49	C/C++ Header	3 KB
cvaux.hpp	2013/12/20 17:49	C/C++ Header	3 KB
cvwimage.h	2013/12/20 17:49	C/C++ Header	3 KB
cxcore.h	2013/12/20 17:49	C/C++ Header	3 KB
cxcore.hpp	2013/12/20 17:49	C/C++ Header	3 KB
cxeigen.hpp	2013/12/20 17:49	C/C++ Header	3 KB
cxmisc.h	2013/12/20 17:49	C/C++ Header	1 KB
highgui.h	2013/12/20 17:49	C/C++ Header	3 KB
ml.h	2013/12/20 17:49	C/C++ Header	3 KB

图 2-39　OpenCV1 的头文件

opencv2 文件夹中包含的是 OpenCV 2 系列的头文件，该文件夹包含了整个 OpenCV 2 系列的精华，如图 2-40 所示。OpenCV 2 系列相比于 OpenCV 1 系列丰富了很多内容，存放的格式也更加规范，相似功能的头文件以一个文件夹的形式存放在 opencv2 中。

calib3d	2014/4/15 17:08	文件夹	
contrib	2014/4/15 17:08	文件夹	
core	2014/4/15 17:08	文件夹	
features2d	2014/4/15 17:08	文件夹	
flann	2014/4/15 17:08	文件夹	
gpu	2014/4/15 17:08	文件夹	
highgui	2014/4/15 17:08	文件夹	
imgproc	2014/4/15 17:08	文件夹	
legacy	2014/4/15 17:08	文件夹	
ml	2014/4/15 17:08	文件夹	
nonfree	2014/4/15 17:08	文件夹	
objdetect	2014/4/15 17:08	文件夹	
ocl	2014/4/15 17:08	文件夹	
photo	2014/4/15 17:08	文件夹	
stitching	2014/4/15 17:08	文件夹	
superres	2014/4/15 17:08	文件夹	
ts	2014/4/15 17:08	文件夹	
video	2014/4/15 17:08	文件夹	
videostab	2014/4/15 17:08	文件夹	
opencv.hpp	2013/12/20 17:49	C/C++ Header	3 KB
opencv_modules.hpp	2014/4/15 17:04	C/C++ Header	1 KB

图 2-40　OpenCV 2 的头文件

这些头文件的功能和用处如下：

（1）calib3d。

calib3d 的全称是 Calibration（校准）+ 3D，该模块的主要作用是利用摄像机进行图像校准和三维重构，其中包括的基本内容有多视觉几何法、单个立体摄像头标定、物体姿态估计、立体相似性算法、3D 信息的重建等。

（2）contrib。

contrib 的全称是 Contributed（贡献）/Experimental（实验）Stuf，这个模块是一个最新贡献但是却仍然在测试中的模块，对于 OpenCV 2.4.9 来说是测试不成熟的模块，但是对于 OpenCV 3 系列就相对成熟一些。这个库中包含了人脸识别、立体匹配等技术。

（3）core。

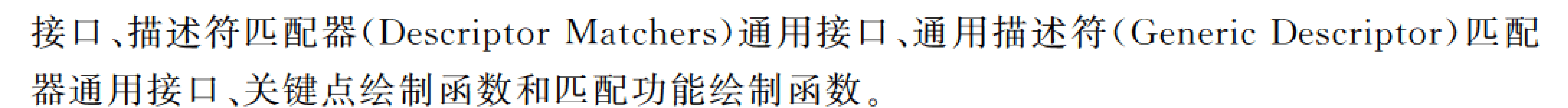
core 是核心功能模块，可以说是 OpenCV 使用率最高的模块。core 模块包含了 OpenCV 基本的数据结构、动态数据结构、绘图函数、数组操作相关函数、辅助功能与系统函数、与 OpenGL 之间的相互协同工作。

（4）imgproc。

imgproc 的全称是 Image（图像）+Process（处理），即为图像处理模块，主要包含了线性和非线性的图像滤波函数、图像的几何变换函数、图像转换函数、直方图相关函数、结构分析函数、形状描述的函数、运动分析函数、对象跟踪函数、特征检测函数、目标检测等函数。

（5）features2d。

features2d 全称就是 Feature+2D，即 2D 功能框架，其主要包含以下内容：特征检测和描述、特征检测器（Feature Detectors）通用接口、描述符提取器（Descriptor Extractors）通用



接口、描述符匹配器(Descriptor Matchers)通用接口、通用描述符(Generic Descriptor)匹配器通用接口、关键点绘制函数和匹配功能绘制函数。

(6) flann。

flann 的全称是 Fast Library for Approximate Nearest Neighbors,是高维的近似近邻快速搜索算法库,其主要内容为快速近似最近邻搜索、聚类等。

(7) gpu。

gpu 模块是将 OpenCV 与计算机中 GPU 连接在一起的接口,也是 OpenCV 使用 CUDA 加速时必不可少的模块。

(8) highgui。

highgui 模块是高层 GUI 图形用户界面模块,也是做图像处理必不可少的模块,这个模块包含了图像视频等的输入和输出、视频捕捉、图像和视频的编码解码、图形交互界面接口等内容。

(9) legacy。

legacy(遗赠)模块是一些已经废弃的代码库,保留下来用于向下兼容,其包含如下功能:运动分析、期望最大化、直方图、平面细分(C API)、特征检测和描述(Feature Detection and Description)、描述符提取器(Descriptor Extractors)的通用接口、通用描述符(Generic Descriptor Matchers)的常用接口、匹配器等。

(10) ml。

ml 的全称是 Machine Learning(机器学习),基本上是统计模型和分类算法,主要包含以下内容:

统计模型(Statistical Models)

一般贝叶斯分类器(Normal Bayes Classifier)

K-近邻(K-Nearest Neighbors)

支持向量机(Support Vector)

决策树(Decision Trees)

提升(Boosting)

梯度提高树(Gradient Boosted Trees)

随机树(Random Trees)

超随机树(Extremely Randomized Trees)

期望最大化(Expectation Maximization)

神经网络(Neural Networks)

MLData

(11) nonfree。

nonfree 就如同其字面意思,为非免费。OpenCV 作为开源代码供大家使用,但是 OpenCV 的开发过程也牵涉到一些专利问题,而此专利的内容就是后面开发 CUDA 加速的 GPU 相关的内容。

(12) objdetect。

objdetect 全称是 Object(目标)+ Detection(检测),即为目标检测模块,主要的用途是将图像中的特征检测出来,如图 2-41 所示,其主要包含了 Cascade Classification(级联分类)

和 Latent SVM 这两个部分。

(13) ocl。

ocl 全称是 OpenCL-accelerated Computer Vision，即利用 OpenCL 加速计算机视觉处理的模块。

OpenCL（全称 Open Computing Language，开放运算语言）是第一个面向异构系统通用目的的并行编程的开放式、免费标准，也是一个统一的编程环境，便于软件开发人员为高性能计算服务器、桌面计算系统、手持设备编写高效轻便的代码，而且广泛适用于多核心处理器（CPU）、图形处理器（GPU）、Cell 类型架构以及数字信号处理器（DSP）等其他并行处理器，在游戏、娱乐、科研、医疗等各个领域都有广阔的发展前景。



图 2-41　objdetect 效果展示

(14) photo。

photo 的全称是 Computational Photography，主要功能是图像的修复和图像去噪两部分。

其中包含的算法有如下几项：

```
CV_EXPORTS_W void inpaint
CV_EXPORTS_W void fastNlMeansDenoising
CV_EXPORTS_W void fastNlMeansDenoisingColored
CV_EXPORTS_W void fastNlMeansDenoisingMulti
CV_EXPORTS_W void fastNlMeansDenoisingColoredMulti
```

其中 inpaint 函数是图像去水印的算法，而后边的四个都是在各种场合下适用的非局部均值去噪算法。

(15) stitching。

stitching 的全称是 images（图像） stitching（连接），即图像拼接模块，其主要功能如下：拼接流水线、特点寻找和图像匹配、估计旋转、自动校准、图片歪斜、接缝估测、曝光补偿、图片混合。

(16) superres。

superres 的全称是 Super Resolution，即超级分辨率技术相关功能模块。

(17) ts。

ts 的全称是 OpenCV 测试代码，没有研究的价值，这里不做细究。

(18) video。

video 正如其字面意思，是视频分析组件，该模块包括运动估计、背景分离、对象跟踪等视频处理相关内容。

(19) videostab。

videostab 的全称是 Video（视频）Stabilization（稳定），即视频稳定模块，该模块主要用来介绍如何在做视频处理的过程中尽量保持视频稳定性[5]。

以上就是 OpenCV 的所有模块，相信了解了这些再有的放矢，学习效率会大大提高。

2.5 OpenCV 环境搭建中常见的问题及解决方案

搭建 OpenCV 环境时经常会遇到各种各样的问题，可能是兼容性问题，也有可能是路径等问题。虽然解决起来没有很大难度，但是解决的过程中会浪费大量的时间和精力。本节将把常见的 OpenCV 环境搭建容易出现的问题一一列出来，分析原因，并给出解决方案，希望可以帮助读者减少在环境搭建时浪费的时间。

2.5.1 无法启动程序

“无法启动程序”通常会以如图 2-42 所示的方式展现出来，无法启动程序后边会跟一个路径，顺着这个路径查找后会发现这个文件夹内什么都没有。

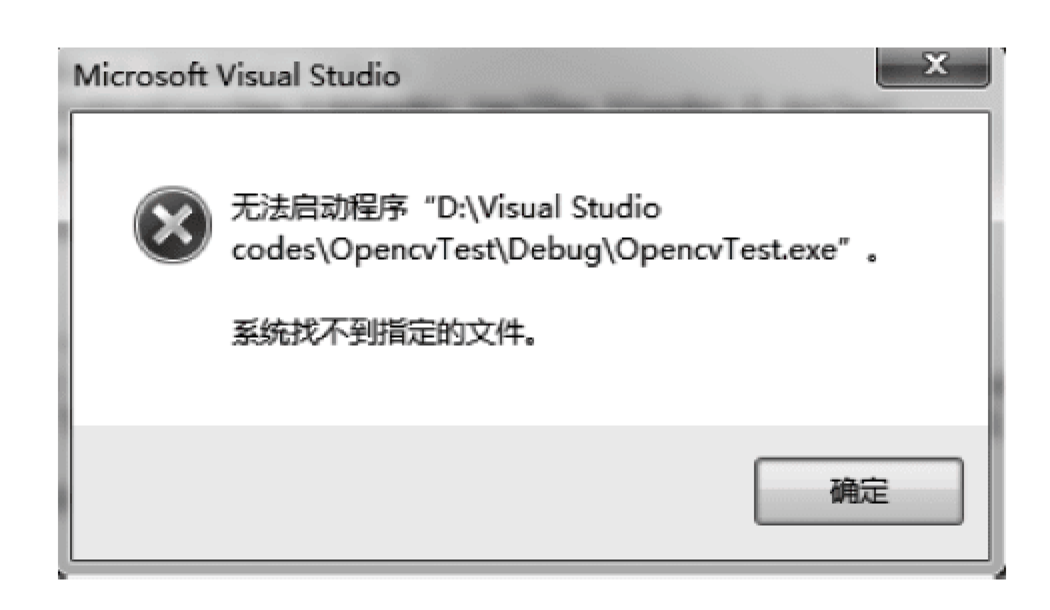


图 2-42 常见错误——无法启动程序

导致这个问题的原因可能是计算机装过高版本的 Visual Studio，但是因为一些原因需要卸载，在没有卸载干净的情况下，就重新装了低版本的 Visual Studio。不同的架构导致了这个问题。

同一个安装错误也会导致如下所示的另一个错误：

LINK：fatal error LNK1123：转换到 COFF 期间失败：文件无效或损坏。

解决方案：

方案一，在之前配置过程中如果没有将“嵌入清单”选项改成“否”，则需要重新修改。

方案二，如果“嵌入清单”选项已经改为“否”之后仍会出现这个情况，那么就需要找到计算机中的 cvtres. exe 文件。cvtres. exe 是资源转换器程序，Windows 下的资源是用脚本来描述的，例如资源中定义的菜单、对话框等，这种脚本表示的资源后缀名是. res 和 rc. exe。对脚本进行处理，过程类似于代码的编译——脚本被处理成二进制格式，处理后的二进制文件要想链接到 exe 中供应用程序使用，要使用 cvtres 将上面的二进制文件转换到通用对象文件格式(COFF)的资源文件，这样 link 才能链接到程序中[6]。

要找到 cvtres. exe 文件，需要在整台计算机中搜索该文件，这个过程需要的时间有些长，搜索的结果正常会有 2～3 个 cvtres. exe 文件，如图 2-43 所示，但实际上会出现很多个，VS 会在每一次安装生成这些文件，但是在每次卸载时都没办法自动地清除这些文件，才导致了这个情况。

解决方案是查看每一个 cvtres. exe 文件的属性，查看每个 cvtres. exe 文件的版本号，如图 2-44 所示。

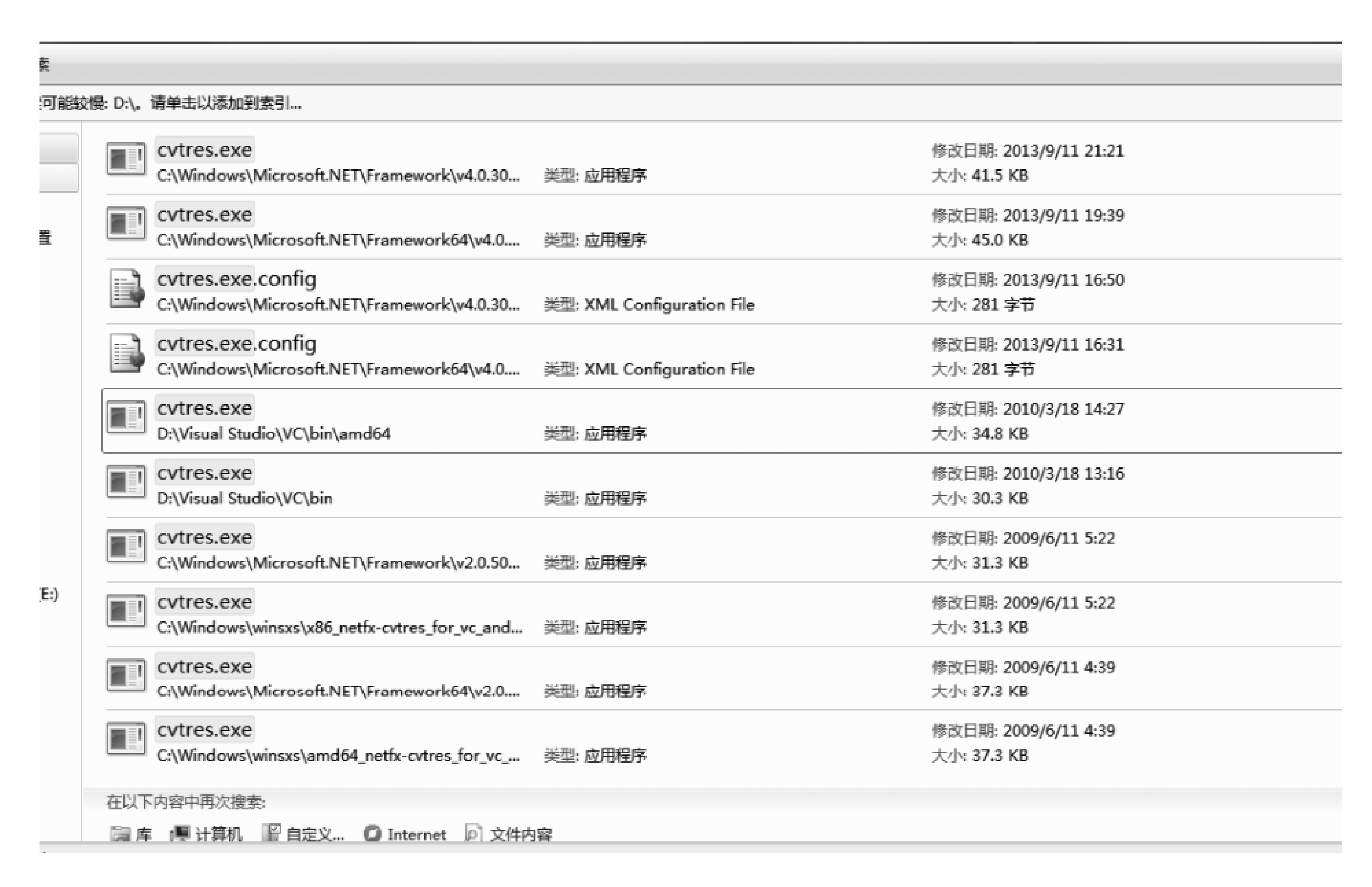


图 2-43 常见错误——查找 cvtres. exe 文件

查看产品名称，如果产品名称是图 2-44 中的 Microsoft. NET Franmework 或 Microsoft Visual Studio 2005 或者是其他，不要改动。如果文件是如图 2-45 所示的 Microsoft Visual Studio 2010，则从这些文件中再查看版本号，将文件中较旧的版本删除或者重命名即可。

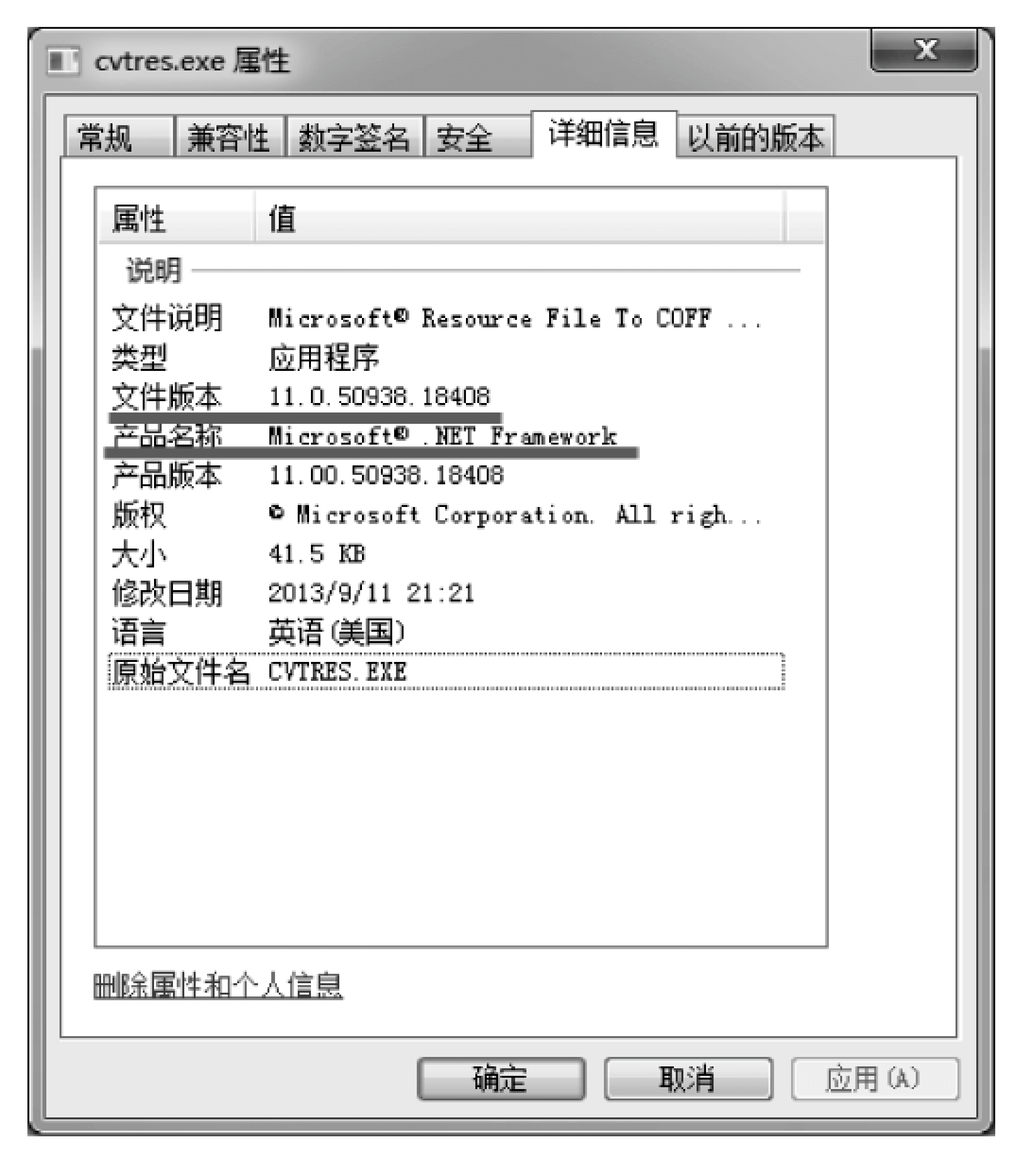


图 2-44 常见错误——cvtres. exe 的版本型号



图 2-45 常见错误——cvtres. exe 版本型号的查询

所有可以删除的文件也可以用文件重命名的方式修改，而且建议使用重命名的方式进行修改。例如上面的 cvtres. exe 文件，为了让其不正常运行，将名字改为“cvtres(删除).

exe”或者是其他名字即可。这样相当于删除了这个文件，并且假如文件删除错误，仍然可以搜索 cvtres. exe 将该文件找出来，如图 2-46 所示。

图 2-46　以重命名方式修改 cvtres. exe 文件

2.5.2　文件缺少 MSVCP110D. dll

“文件缺少 MSVCP110D. dll”可能是因为曾经装过 VS 2012，卸载后使用 VS 2010，运行程序时会出现的错误，这种错误通常会以如图 2-47 和图 2-48 的形式表现出来。

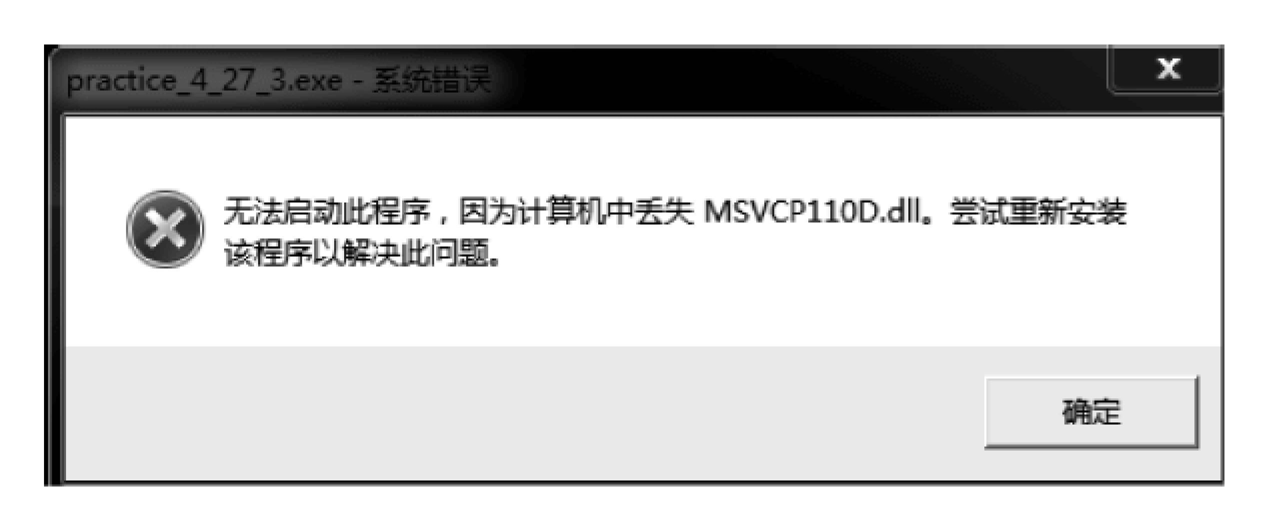


图 2-47　常见错误——缺失 MSVCP110D. dll

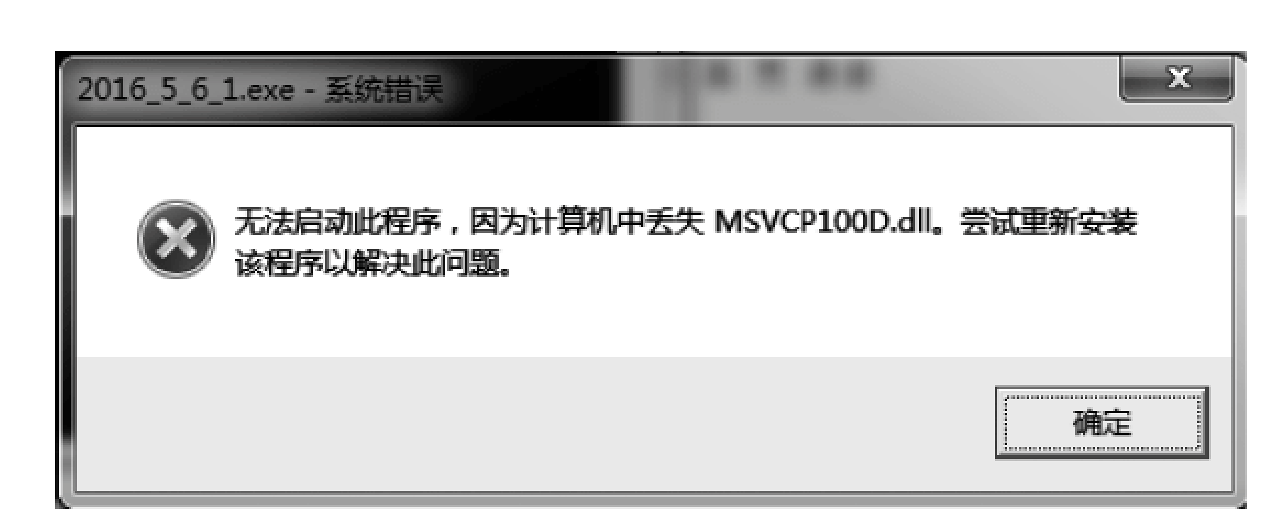


图 2-48　常见错误——缺失 MSVCP100D. dll

遇到这两种情况比较麻烦，下面给出两种解决方案，供读者尝试。

方案一：

先给出缺少 MSVCP110D. dll 的解决方案，缺少 MSVCP100D. dll 与缺少 MSVCP110D. dll

的解决方法是一样的。

首先要到 Microsoft 官网下载一个插件，下载地址如下：

http://www.microsoft.com/zh-CN/download/details.aspx?id=30679

之后选择“中文(简体)”选项，并下载，如图 2-49 所示。

Visual C++ Redistributable for Visual Studio 2012 Update 4

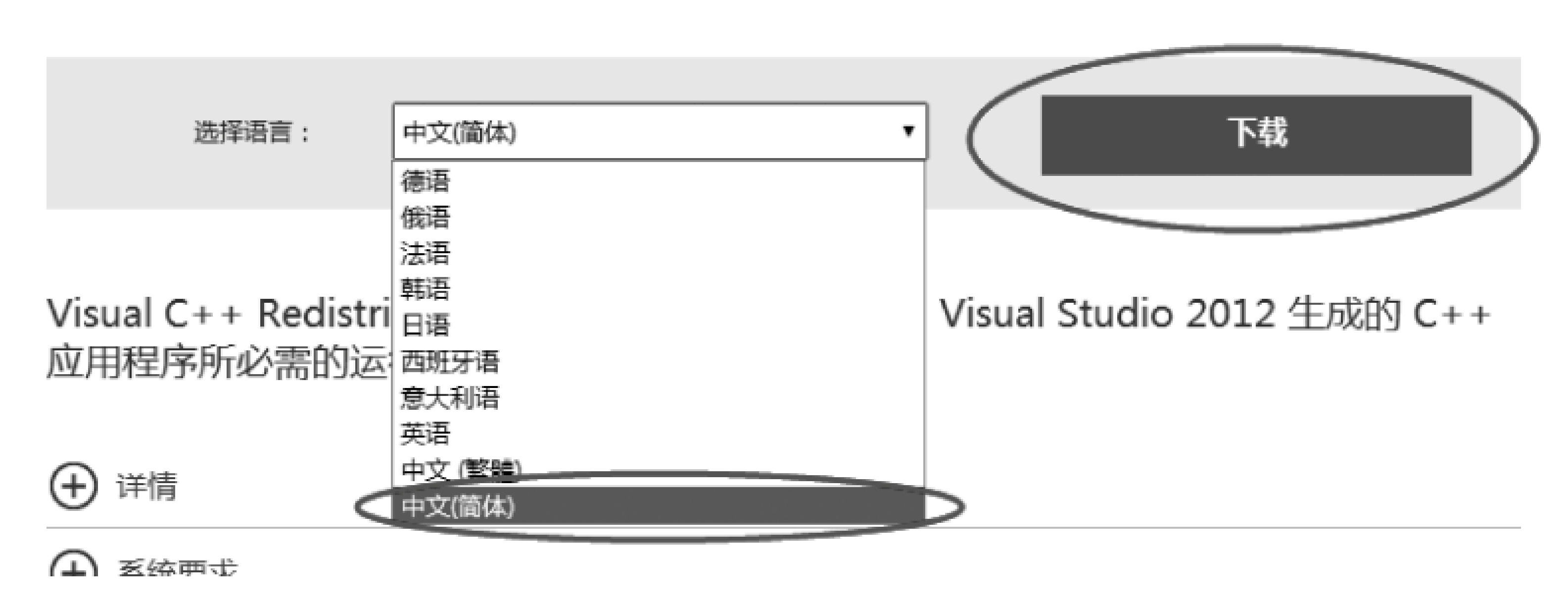


图 2-49　MSVCP110D.dll 下载

下载选项要根据计算机的系统选择 x64 或者 x86，这里不是指编写 x86 或是 x64 的程序，示例中的计算机是 64 位系统，因此选择的也是 x64，如图 2-50 所示。



图 2-50　MSVCP110D.dll 型号选择

下载完成之后安装即可，正常的安装界面如图 2-51 所示，如果计算机曾经安装过这个插件，单击“修复”按钮即可，如图 2-52 所示。

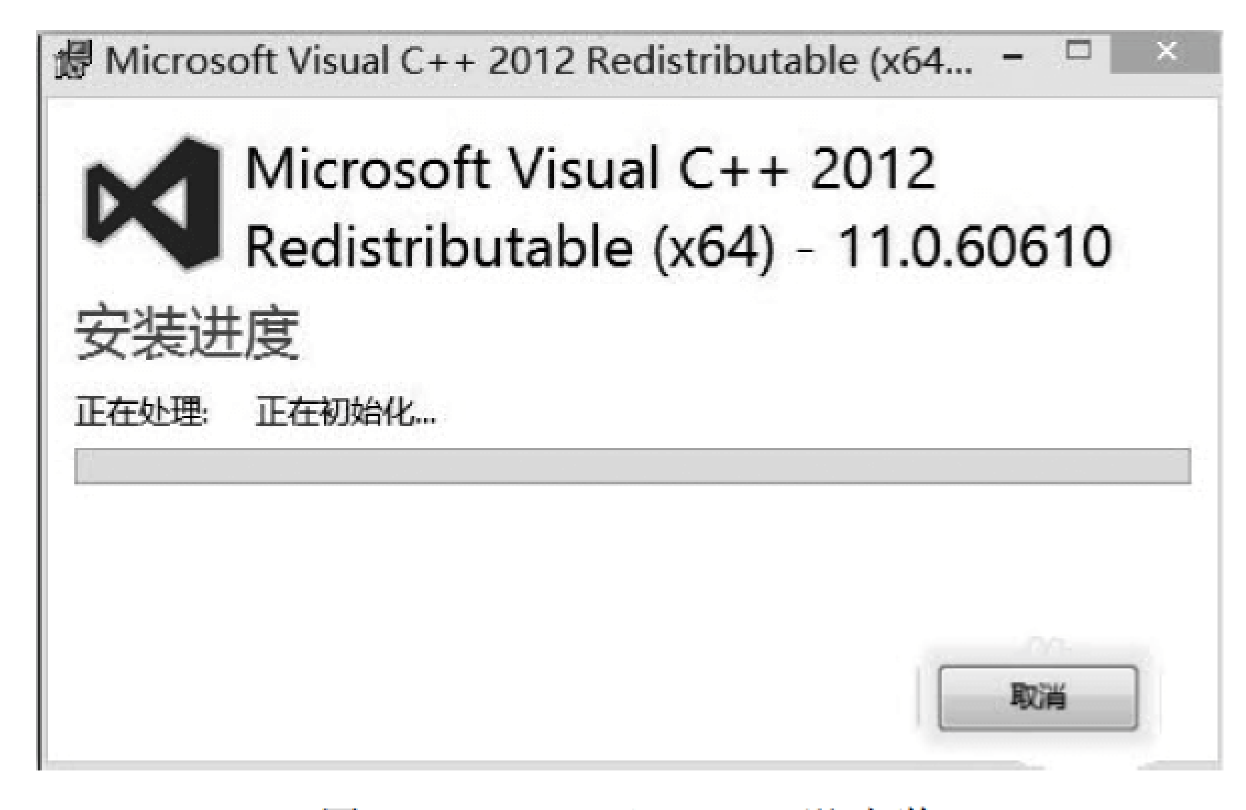


图 2-51　MSVCP110D.dll 安装



图 2-52　MSVCP110D. dll 修复

安装成功的界面如图 2-53 所示。

缺少 MSVCP100D. dll 的解决方案也是一样的，在该网站上寻找 C++2010 版本，按上述步骤安装即可。

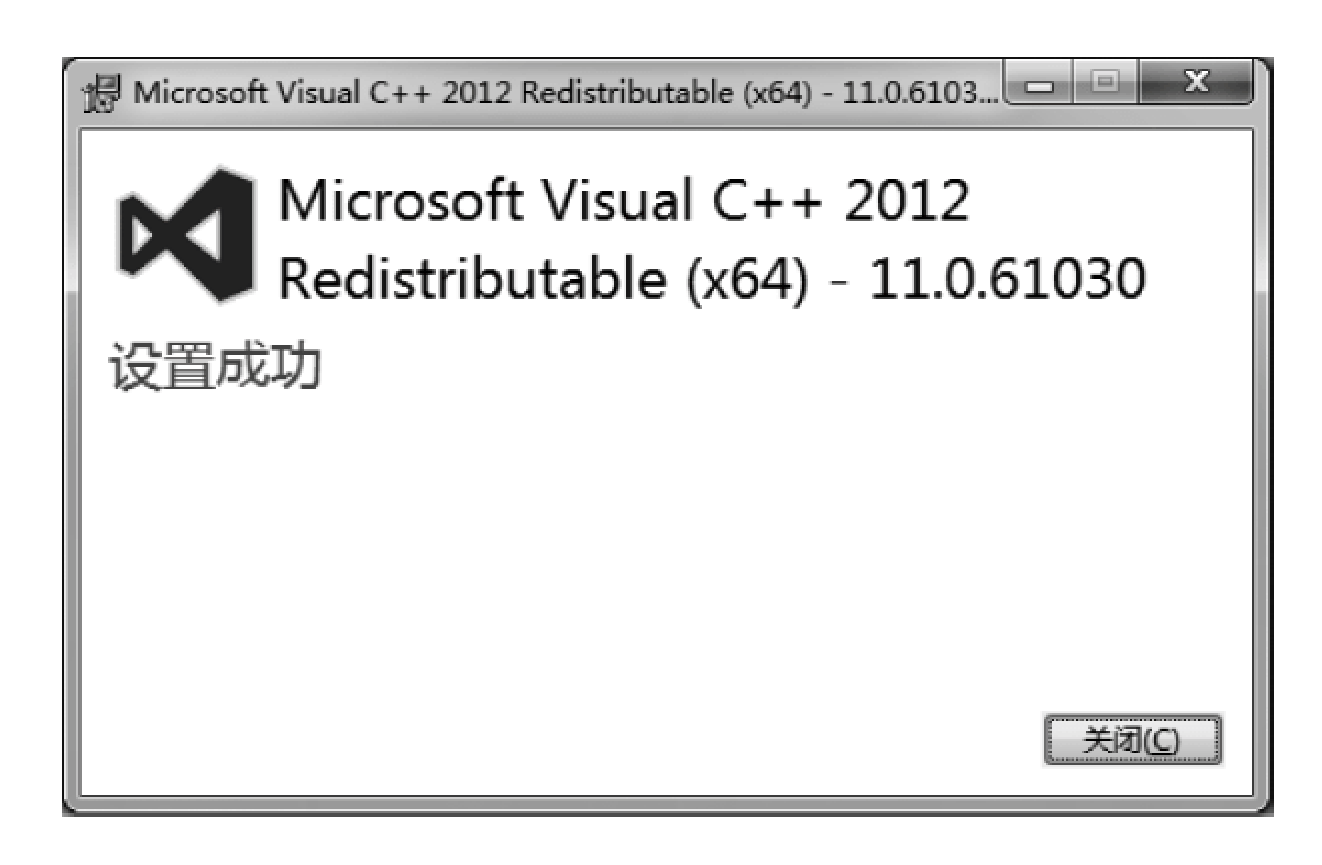


图 2-53　MSVCP110D. dll 安装成功

方案二：

直接去下载 MSVCP110D. dll 文件，如果没有合适的网址，下面给出一个可以供读者参考的下载网址：

```
www.zhaodll.com
```

下载好 MSVCP110D. dll 之后，将这个文件复制到 C 盘中的 System32 或 SysWOW64 文件夹中，其路径如下所示：

```
C:\Windows\System32
C:\Windows\SysWOW64
```

复制的原则是：如果计算机系统是 32 位的就复制到 System32 文件夹中，如果计算机系统是 64 位的就复制到 SysWOW64 文件夹中。

复制完成之后单击“开始”→“运行”命令，之后输入 regsvr32 msvcp110d. dll，如图 2-54

所示，完成之后不论是什么结果，都重启计算机，再回到 VS 中，重新运行之前的程序，就不会遇到这个问题了。



图 2-54　注册 msvcp110d.dll

2.5.3　Cannot find or open the PDB file

在测试的过程中可能会出现如图 2-55 所示的 Cannot find or open the PDB file 错误。



图 2-55　常见错误——Cannot find or open the PDB file

这种情况一般是第一次将 OpenCV 库放在 VS 2010 中运行时会出现。如果选择 Release，反而不出现这个错误，这是因为每次在使用时都需要联网下载一些字体文件，这个问题的解决方案如下。

（1）确保自己的计算机处于联网状态，没有网络是没办法下载的。

（2）在 VS 2010 最上方的工具栏中选择“工具”→“选项”命令，如图 2-56 所示。

（3）进入“选项”界面之后找到“调试”选项，选择“调试”中的“符号”选项，如果第一次出现这类问题会与图 2-57 所示一致。在右侧选中“Microsoft 符号服务器”复选框，单击“确定”按钮。

（4）再次单击“确定”按钮会回到主界面。这时候需要随意运行一个小程序，让 VS 2010 从服务器上下载这些字体符号，选择的程序可以是 2.3.9 节中的环境测试的小程序，如图 2-58 所示。编写完程序后，在 Debug 下选择 x64 平台编译，最后单击“运行”按钮或者按 F5 键运行程序即可。

（5）这次运行程序的时间相对较长，因为这次不仅仅要编译并运行程序，还要将程序编译过程中需要的符号从服务器下载下来，因此需要更多的时间。

（6）下载完成之后，程序即可正常运行。这时需要修改一下符号的来源位置，如果仍然选择从“服务器下载”，那么每次运行都需要消耗下载符号的时间。可以将符号来源选择为已经下载好的本地文件，这样可以避免每次运行重新下载，解决方案如下：



图 2-56 Cannot find or open the PDB file 解决步骤第一步



图 2-57 Cannot find or open the PDB file 解决步骤第二步

在下载完所有的符号，程序可以正常运行一次之后，按照步骤(3)，找到“符号”选项。取消选中“Microsoft 符号服务器”复选框，同时复制下方存放符号的文件夹路径，如图 2-59 所示。

图 2-58 Cannot find or open the PDB file 解决步骤第三步

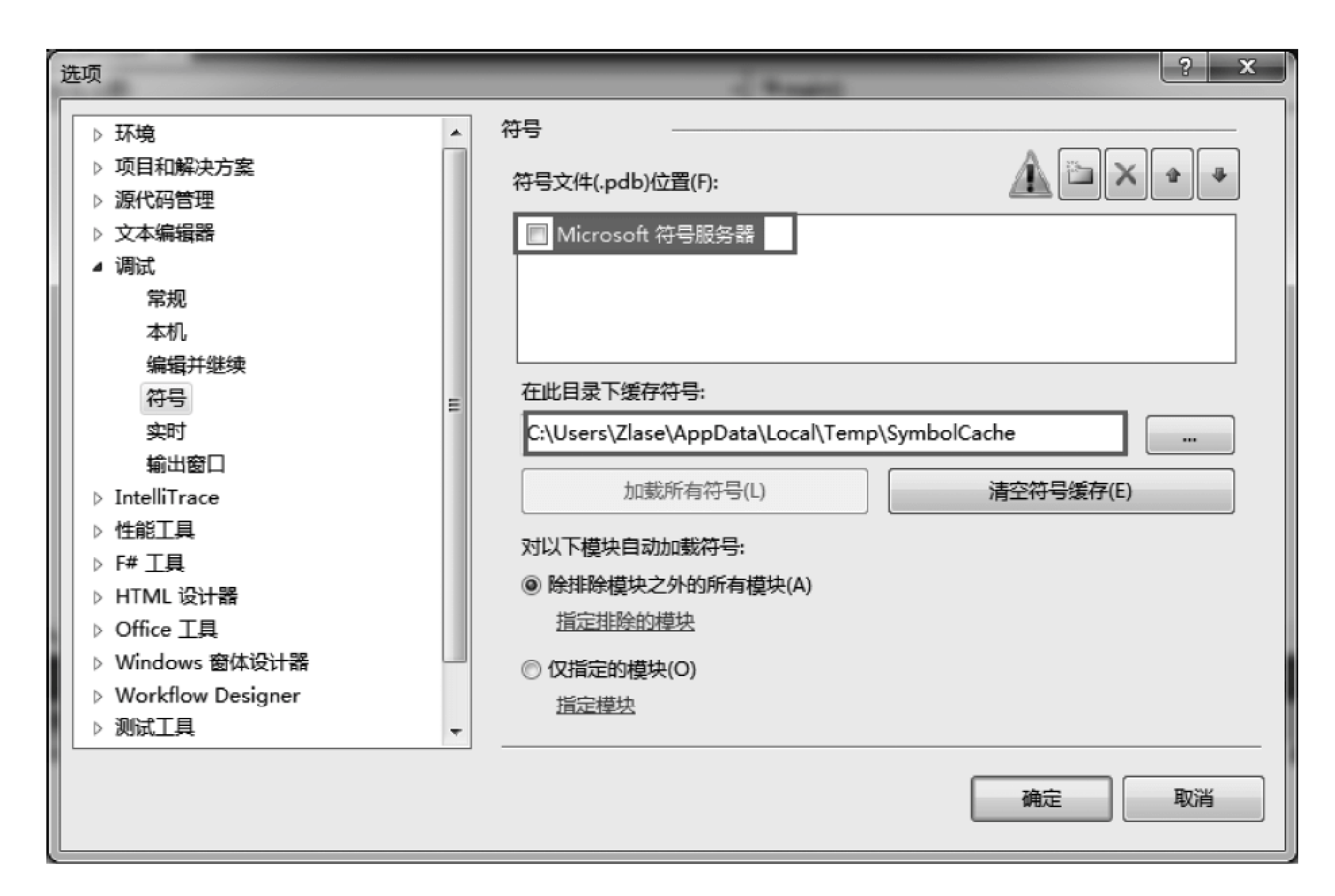


图 2-59 Cannot find or open the PDB file 解决步骤第四步

在完成这些之后单击右上角的小文件夹图标(即“新增”选项),此时下面会出现一个路径栏,将之前复制下来的路径粘贴到这里,如图 2-60 所示,复制完成之后单击“确定”按钮即可回到主界面。

(7) 再次运行程序,如果这次程序运行不出现错误,那么以后运行其他程序也不会出现这样的错误了。

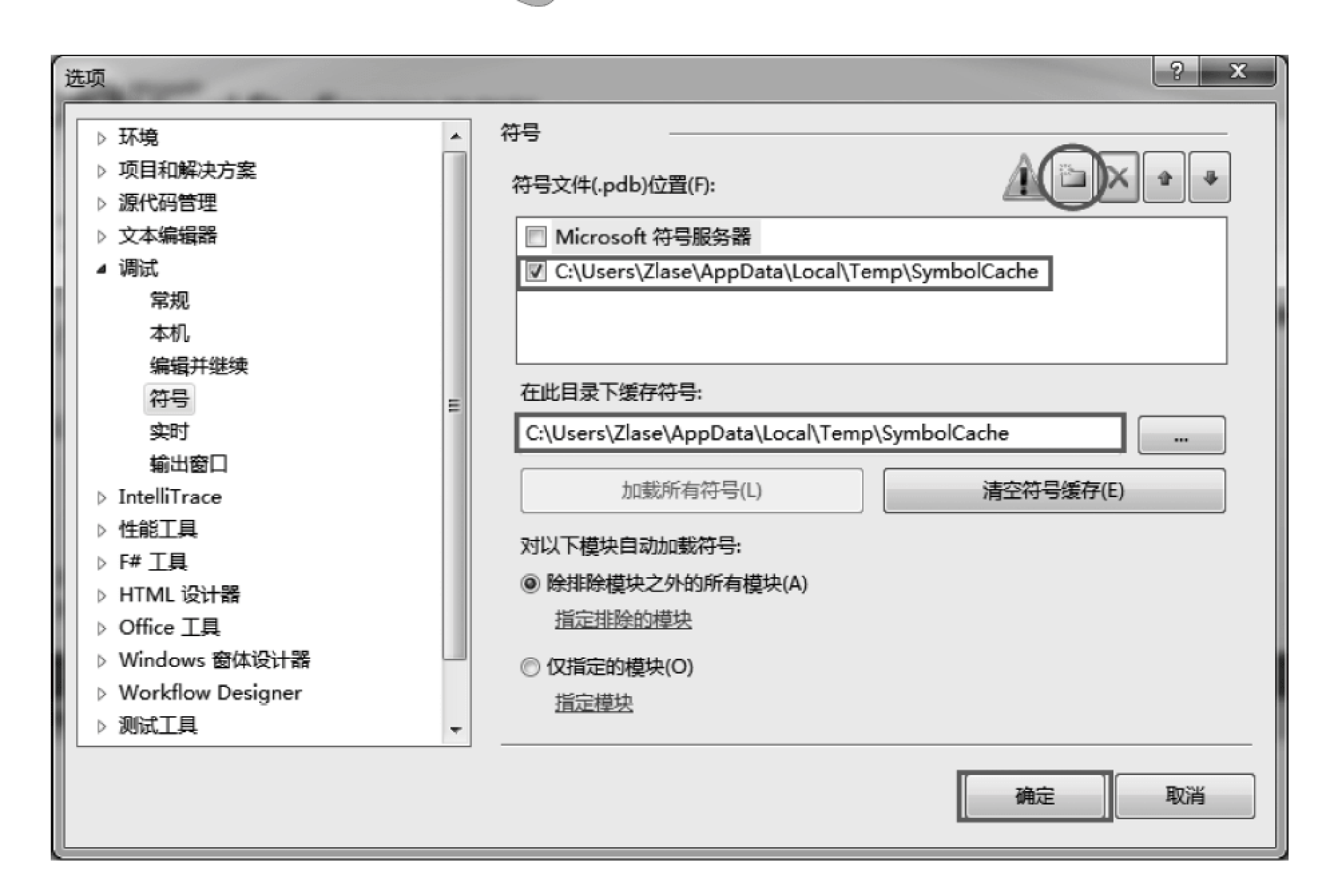


图 2-60　Cannot find or open the PDB file 解决步骤第五步

2.5.4　文件缺少 tbb_debug. dll

"文件缺少 tbb_debug. dll"tbb 库与 OpenCV 没有太大的关系,而且在后面的 CUDA 驱动 GPU 做并行处理的过程中已经安装了 tbb,所以这个问题不是很常见。

文件中缺少 tbb_debug. dll 的解决方案如下。

首先下载一个 tbb41_20130314oss_win. rar 文件,将其解压后的 bin 文件夹中所有的文件复制到:

```
D:\opencv\build\common\tbb
```

第二步在环境变量(也就是之前 2.3.2 节介绍的 path)中添加:

```
D:\opencv\build\common\tbb\ia32\vc10
```

即可解决这个问题。

2.5.5　应用程序无法启动 0xc000007b

"应用程序无法启动 0xc000007b"这个问题不止出现在 OpenCV 编译程序时,编译其他非 OpenCV 程序时也可能会出现这个情况,所以解决方案也五花八门,下面将列举可行性较强的方案供读者参考:

(1) lib 文件包含问题。

如果是 lib 文件包含错误,就说明之前的配置过程出现了问题。检查的方式是建议从 2.3.3 节开始重新配置。

错误可能出现的原因有如下两种:

- 路径添加错误,配置 OpenCV 路径时在 Path 中没有将所有要添加的路径都添加进

去，或者是添加后没有用“；”隔开。

- lib 文件放入位置错误，正常的 32 位系统，要将 x86 中的 lib 文件放在 System32 文件夹中，而 64 位系统需要将 x64 文件夹的 lib 文件放在 SysWOW64 文件夹中。

（2）DirectX 9.0 组件损坏。

这个方案可能与本次 OpenCV 的配置没什么直接关系，但是同样作为驱动 GPU 实现图像显示的插件，也可能是这个原因导致的，解决方案是重新安装或修复 DirectX 系列文件。

因为不确定是否仅有一个 DirectX 9.0 被损坏，所以选了一个可以测试所有 DirectX 系列文件的软件：DirectX 9.0 c 软件，其 Logo 和安装包的外貌如图 2-61 所示。

下面将提供两款 DirectX 修复工具：一款是用在 Windows 7 上的，另外一款是用在 Windows 8 和 Windows 10 上的，这两款软件都在同一个文件夹中。读者可以去 DirectX 官网自行下载或者按照如下所示的百度网盘链接进行下载。

DirectX 修复工具链接：

链接：http://pan.baidu.com/s/1jHN6m5w 密码：blq9

下载并解压完成之后会出现如图 2-62 所示的文件，左侧为压缩包，右侧两个是解压之后的文件。



图 2-61　DirectX 9.0 c Logo

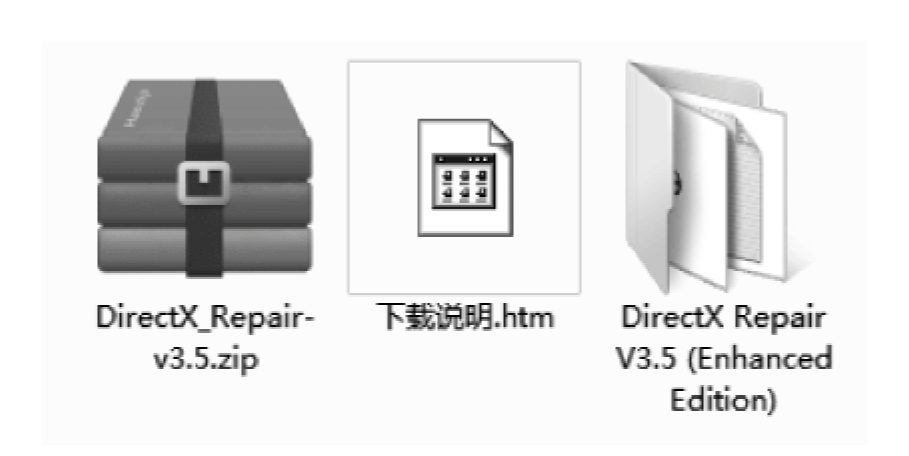


图 2-62　DirectX_Repair 文件下载

打开解压后的文件夹，其中有两个.exe 文件，如图 2-63 所示，上方的.exe 文件是在 Windows 7 系统下使用的，下边的.exe 文件是在 Windows 8 和 Windows 10 系统下使用的。

名称	修改日期	类型	大小
Data	2016/6/23 13:04	文件夹	
DirectX Repair.exe	2016/6/23 10:50	应用程序	633 KB
DirectX_Repair_win8_win10.exe	2016/6/23 10:52	应用程序	633 KB
Settings.ini	2016/6/22 17:42	配置设置	1 KB
常见问题解答.txt	2016/6/22 16:44	文本文档	21 KB
更新日志.txt	2016/6/23 10:57	文本文档	21 KB
技术文档.txt	2016/6/22 16:51	文本文档	6 KB
使用说明.txt	2016/6/22 17:43	文本文档	14 KB
致Windows XP用户.txt	2016/6/22 16:55	文本文档	1 KB

图 2-63　DirectX Repair 文件安装包

示例中使用的是 Windows 7 系统，双击上方的.exe 文件，会进入如图 2-64 所示界面，单击“检测并修复”按钮，等待即可。



图 2-64 DirectX Repair 修复过程

如果出现如图 2-65 所示的自动更新 C++库的程序，可以单击“取消”按钮更新，这个更新的过程不会影响后续的修复步骤。

当全部修复完成之后会弹出如图 2-66 和图 2-67 所示的结束窗口。



图 2-65 DirectX Repair 更新 C++库



图 2-66 DirectX Repair 检测结果

2.5.6 找不到头文件

这个问题一般显示为：找不到 xxxxxx.h 文件。

出现这个问题是因为前面环境配置过程的某一步出现了错误，导致程序找不到头文件，可能的原因有以下几点。

1. 路径

例如版本 OpenCV 2.4.9，在 opencv 文件夹中有一个 include 文件夹。但是在正常配置时需要的是 build 文件夹中的 include 文件夹，而不是之前所说的 opencv 文件夹下的 include，所以千万不能搞错路径。



图 2-67 DirectX Repair 检测和修复完成

2. 附加依赖项

属性配置时，在添加 Release 中“附加依赖项”这一步，添加的 lib 文件没有将末尾的 d 去掉，如 Debug 的附加依赖项为 opencv_highgui249d. lib，而在 Release 中应该为 opencv_highgui249. lib。

3. 编程习惯

编写程序的头文件时，需要将头文件的路径也加进去，例如原本包含的头文件为：

```
#include <highgui.h>
```

现在可以换一种写法：

```
#include <opencv2/highgui/highgui_c.h>
#include <opencv2/highgui/highgui.hpp>
```

4. 没有此类头文件

出现“找不到头文件”的错误，也可能是根本没有这种头文件。被调用的头文件不是 OpenCV 或者是系统自带的头文件。

2.5.7 无法打开 lib 文件

这种错误通常会表现为：

```
fatal error LNK1104:无法打开文件 opencv_ml249d.lib
```

这种错误是由于之前配置过程不完全正确而导致的，最可能的原因如下：

在添加附加依赖项时，在填写 opencv_ml249d. lib 时在前边加上了一个空格，即“opencv_ml249d. lib”这样的情况也算添加失败，会报错。

如果不是空格导致错误，建议检查之前进行的 include 配置、库目录配置和附加依赖项配置。

如果上述两种方案都没有解决这个问题，则需要一种比较麻烦的方法。

按照 2.3.3 节的介绍，回到项目的属性配置页面。在“配置属性”中找到“链接器”选项，打开后选择“常规”选项，在右侧可以看到“附加库目录”选项，如图 2-68 所示。

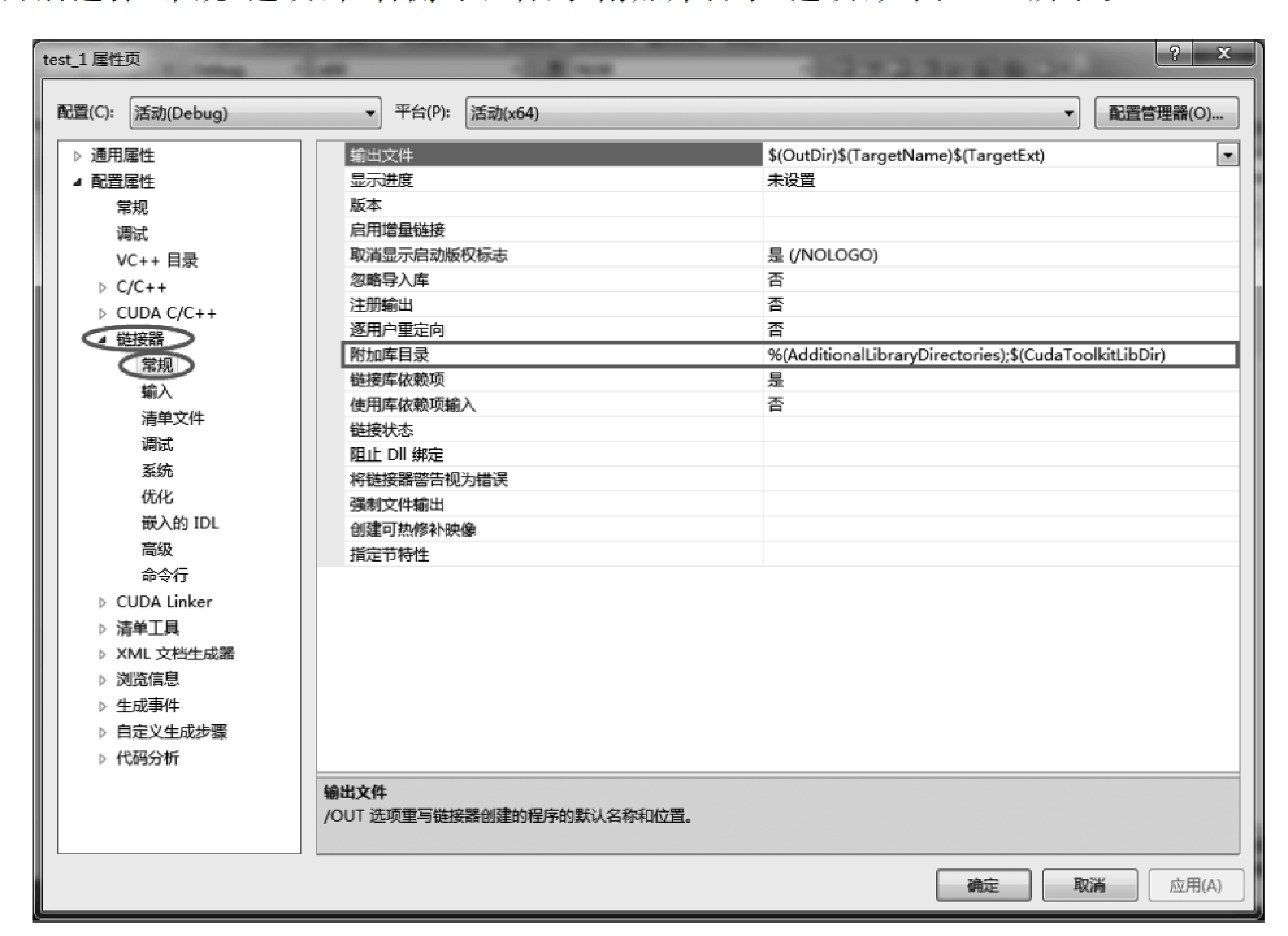


图 2-68 无法打开 lib 文件解决方案第一步

打开“附加库目录”选项，编辑其中的 lib 路径，如图 2-69 所示。



图 2-69 无法打开 lib 文件解决方案第二步

这里添加的路径就是缺失的 lib 文件的路径，缺少的 lib 文件只能到 opencv 文件夹中找，而且添加路径仅能一条一条添加，所以这个方法比较麻烦。

例如前面缺少 opencv_ml249d.lib 文件，现在需要在 opencv 文件夹中找到这个 lib 文件，并添加进去，示例中的 lib 文件的路径为：

```
D:\OpenCV2.4.9\opencv\build\x64\vc10\lib\opencv_photo249d.lib
```

如果还缺失其他文件，也可以通过这种方式进行补充，但是要注意 Debug 和 Release 的 lib 文件是不同的，也需要注意带"d"和不带"d"的区别。

添加完成的附加库目录如图 2-70 所示。



图 2-70 无法打开 lib 文件解决方案第三步

在 Debug 下，添加完附加库目录后，回到属性页面的 Release 界面重新添加一次即可解决这个问题。

2.5.8 指针越界 cv::Exception

这种错误体现为：程序已经成功编译，但是在运行的过程中会出现没有办法载入图片或空指针或指针溢出的情况。如果切换 Debug 和 Release，两者其中一个会出现错误，另外一项是可以正常执行的，如图 2-71 所示。

这种问题不是因为环境搭建出错造成的，这个错误其实是 OpenCV 2.4 系列自身的一个 bug。这个问题的核心是"附加依赖项"的添加问题，在前面配置附加依赖项时需要先配置 Debug 或者 Release 其中之一，然后再配置另外一个，但是 OpenCV 这时候会默认选择先填入的附加依赖项作为调试过程中的 lib 文件。

简单来说，如果先配置好 Debug 后再配置 Release，那么在调试的过程中就会出现 Debug 可以成功但是 Release 报错的情况；反之，如果先配置 Release 再配置 Debug，调试时会出现 Release 成功但 Debug 报错。

通常情况下，严格按照 2.3 节的配置顺序进行配置是不会出现这个问题的，但是在实际

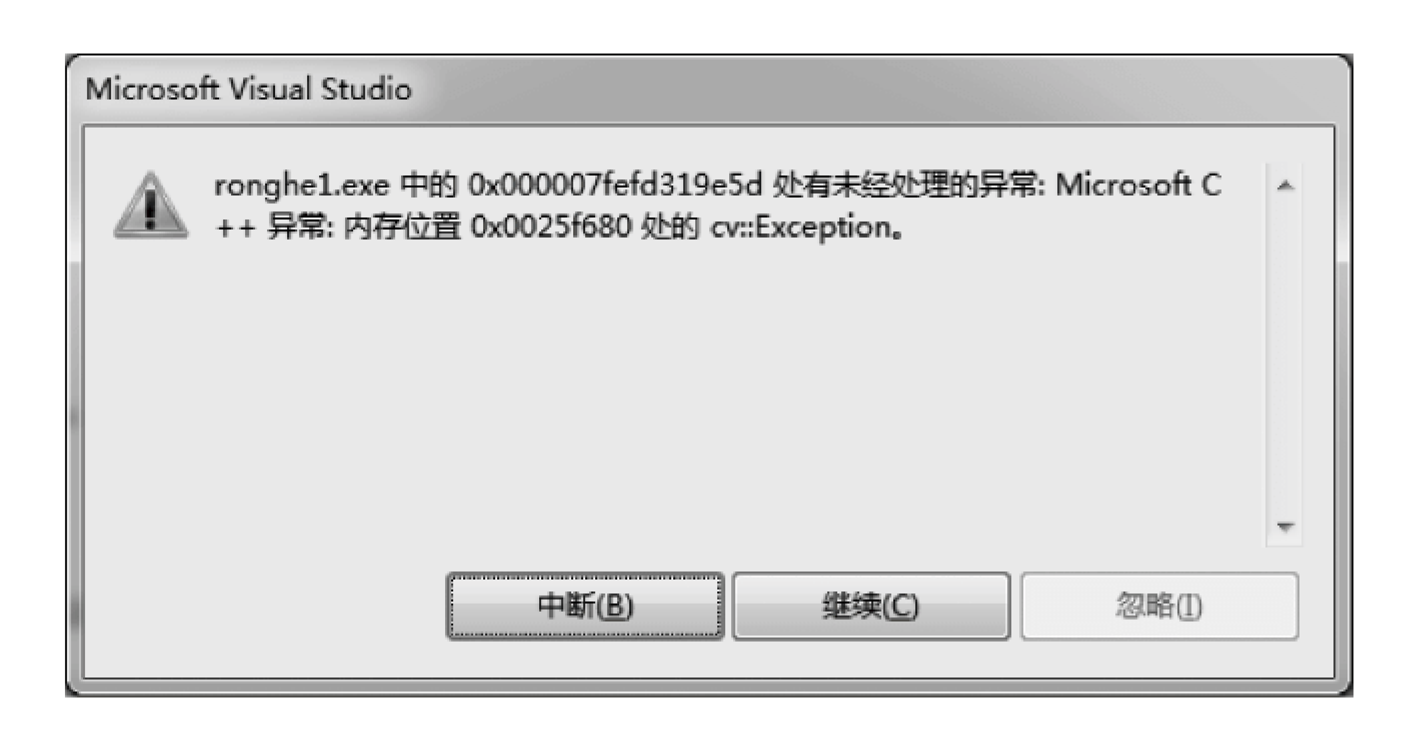


图 2-71 指针越界的错误提示

使用中如果出现这种问题，解决方案如下：

新建一个项目文件，配置好 Debug 后，将程序在 Debug 调试模式下运行一次，这个结果应该是成功的（必须在路径中添加好相应的图片，否则也会出现这个问题），之后再打开属性界面，在 Release 模式下将环境搭建好，搭建好后在 Release 调试模式下运行一次程序，就可以解决这个问题。如果最开始是在 Release 下配置好，然后配置 Debug，就无法在两个调式模式下均顺利运行[7]。

如果用上述方法没有解决该问题，则说明编写的程序本身会导致指针越界，需要修改已编写的程序。

2.5.9 x86 与 x64 类型冲突

在调试程序的过程中可能会出现“模块计算机类型 x64 与目标计算机类型 x86 冲突”的问题。

出现这种问题是因为编译的程序是 x64 的程序，但是使用的是 x86 平台，或是因为环境搭建路径配置时，没有找到正确的 lib 文件。

解决方案：

如果使用 x64 的程序但是使用的是 x86 平台，即 Win32 平台，只要将平台更换成 x64 平台即可，如图 2-72 所示。

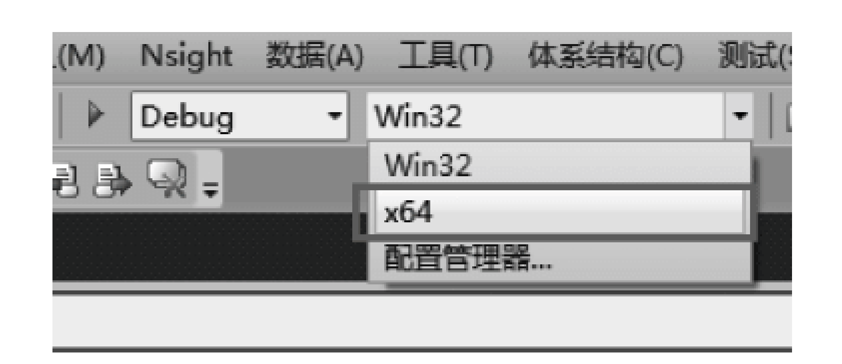


图 2-72 x86 与 x64 类型冲突解决方案——更换平台

如果下拉列表框中没有 x64，则必须新建一个 x64 平台，具体过程请看 2.3.3 节的（4）和（5）两步。

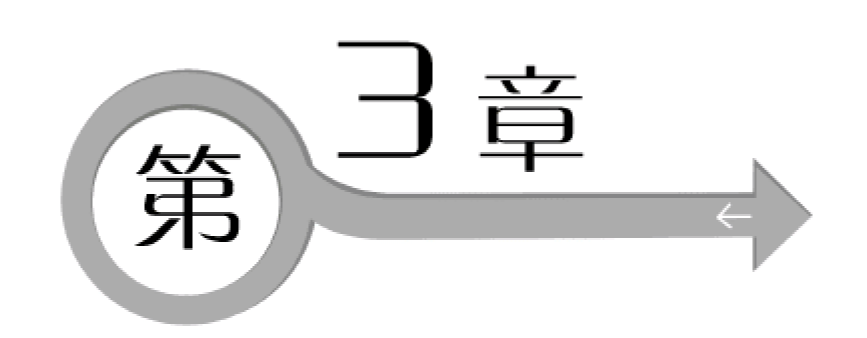
如果用上述方案仍然无法解决问题，那么就是路径配置出现了问题，即没有把 x64 和 x86 正确地加入进去。

2.6 本章小结

本章主要介绍了机器视觉库 OpenCV 的相关知识。2.1 节详细介绍了 OpenCV 的发展历程。2.2 节详细介绍了 OpenCV 基于的 C++开发平台 Visual Studio，并介绍了如何安装 Visual Studio 2010。2.3 节详细介绍了 OpenCV 2.4.9 在 Windows 系统上的搭建过程，并给出一个可以进行环境测试的小程序。2.4 节介绍了 OpenCV 2.4.9 的基本架构，在搭建好环境后可以直接查看这些源码，它可以帮助读者对 OpenCV 架构有一个详细的了解。2.5 节将常见的 OpenCV 环境搭建中出现的问题进行整理，并给出了详细的解决方案，可以帮助读者顺利完成环境搭建。

2.7 参考文献

[1] http://opencv.org/news.html
[2] http://baike.baidu.com/item/Microsoft%20Visual%20Studio%202010?fromtitle=vs2010&fromid=10871755&sefr=ps
[3] http://www.opencv.org.cn/
[4] 毛星云，冷雪飞. OpenCV 3 编程入门[M]. 北京：电子工业出版社，2015.
[5] Bradski G，Kaehler A. Learning OpenCV：Computer Vision in C++ with the OpenCV Library[M]. O'Reilly Media，Inc. 2013.
[6] http://bbs.csdn.net/topics/290013753
[7] https://stackoverflow.com/questions/19925451/unhandled-exception-at-0x7508c41f-in-matrixprojection-exe-microsoft-c-excepti



OpenCV常用函数和应用实例

本章主要介绍 OpenCV 的常用函数、变量以及基本数据结构。在此基础上，给出部分应用示例，详细讲解反向、滤波、双目视觉测深度算法，为灵活使用 OpenCV 打下基础。

3.1 OpenCV 常用函数

本节主要针对 OpenCV 中最常用的，也是最核心的 Mat 类、imread 函数、imshow 函数以及 imwrite 函数进行介绍。

3.1.1 Mat 类

说到 Mat 类就必须要介绍一下 OpenCV 2 系列。在 OpenCV 1 系列的函数库中，函数都是基于 C 接口构建的，使用名为 IplImage 的 C 语言结构体在内存中存储图像，用法类似于 C 语言中的指针，在使用函数之前先为变量开辟内存，使用之后释放内存，在算法复杂度较高的情况下，指针的使用会引起内存混乱。而在 OpenCV 2 系列中，C++的出现带来了全新的 Mat 类，使 OpenCV 能执行自动的内存管理，也就是说，在使用 OpenCV 时不必再进行手动开辟内存、管理内存等烦琐的工作，大大减少了开发者的工作量。自动开辟指的是程序自动根据需要使用的内存空间，合理开辟内存，不会造成资源的浪费，当然如果必要，也可以手动开辟内存空间[1]。

1. Mat 类的定义

官方手册上 Mat 的定义是：Mat 类用于表示一个多维度的单通道或者多通道的稠密数组，能够用来保存实数或复数的向量、矩阵、灰度或彩色图像、立体元素、点云、张量以及直方图等[2]。

在使用的过程中只需要记住两点：

(1) Mat 类用来保存多维度的矩阵；

(2) Mat 类不需要手动开辟内存，也不需要手动释放内存。

2. Mat 的数据结构

Mat 的数据结构主要包括两个部分：Header 和 Pointer。其中 Header 主要包含矩阵

的大小、存储方式、存储地址等信息，而 Pointer 是存储指向像素值的指针。

Mat 类的常见属性如下：

（1）data。

data 是 uchar 型的指针，也就是上述指向矩阵内数据的指针。

（2）dims。

dims 指的是矩阵的维度，如：一维数组 dims=1；二维矩阵 dims=2 等。

（3）rows、cols。

rows 指矩阵的行数，cols 指矩阵的列数。需要注意的是，如果是多通道的彩色图像，rows 和 cols 分别表示的是图像中纵向的像素点数量和横向像素点数量，这与通道数没有关系。

（4）channels。

channels 指矩阵的通道数，如果读入的是图像，也指图像的通道数。通常单通道的图像为灰度图像，彩色图像有三通道的 RGB（Red、Green、Blue）图像和四通道的 RGBA（Red、Green、Blue、Alpha）图像。

（5）type。

type 表示矩阵中元素的类型以及矩阵的通道数，属于预定义的常量，其命名的规则为[3]：

CV_＋位数＋数据类型＋通道数

如：CV_8UC1 表示 8 位、unsigned integer 无符号型、单通道；

CV_16SC2 表示 16 位、singed integer 有符号整数、双通道；

CV_64FC4 表示 64 位、float 浮点型、四通道。

其具体的型号如表 3-1 所示。

表 3-1　type 常量

CV_8UC1	CV_8UC2	CV_8UC3	CV_8UC4
CV_8SC1	CV_8SC2	CV_8SC3	CV_8SC4
CV_16UC1	CV_16UC2	CV_16UC3	CV_16UC4
CV_16SC1	CV_16SC2	CV_16SC3	CV_16SC4
CV_32SC1	CV_32SC2	CV_32SC3	CV_32SC4
CV_32FC1	CV_32FC2	CV_32FC3	CV_32FC4
CV_64FC1	CV_64FC2	CV_64FC3	CV_64FC4

（6）depth。

depth 指矩阵中元素一个通道内的数据类型，这个和上面的 type 型类似，相比于 type 更加简单。depth 常用常量如表 3-2 所示。

表 3-2　depth 常量

CV_8U	CV_8S	CV_8F
CV_16U	CV_16S	CV_8F
CV_32U	CV_32S	CV_32F
CV_64U	CV_64S	CV_64F

（7）elemSize。

elemSize 是指矩阵中一个元素占用的字节数，其值为：

$$elemSize = 通道数 \times 位数 \div 8$$

如：CV_64FC4；elemSize＝4×64÷8＝32bytes。

（8）elemSize1。

elemSize1 指矩阵元素一个通道占用的字节数，相比于 elemSize 仅少了通道数，如：CV_64FC4；elemSize1＝64÷8＝8bytes。

下面的程序详细示范了 Mat 属性的使用方法及效果：

```
#include <opencv2/opencv.hpp>
#include <iostream>
usingnamespace cv;
usingnamespace std;

void main()
{
    Mat img(3, 4, CV_16UC4, Scalar_<uchar>(1, 2, 3, 4));
    cout << img << endl;
    cout <<"dims:"<< img.dims << endl;
    cout <<"rows:"<< img.rows << endl;
    cout <<"cols:"<< img.cols << endl;
    cout <<"channels:"<< img.channels() << endl;
    cout <<"type:"<< img.type() << endl;
    cout <<"depth:"<< img.depth() << endl;
    cout <<"elemSize:"<< img.elemSize() << endl;
    cout <<"elemSize1:"<< img.elemSize1() << endl;
    getchar();
}
```

该程序定义了一个 3 行 4 列 4 通道 16 位 unchar 型矩阵，存放的数值依次为 1、2、3、4。运行结果如图 3-1 所示。

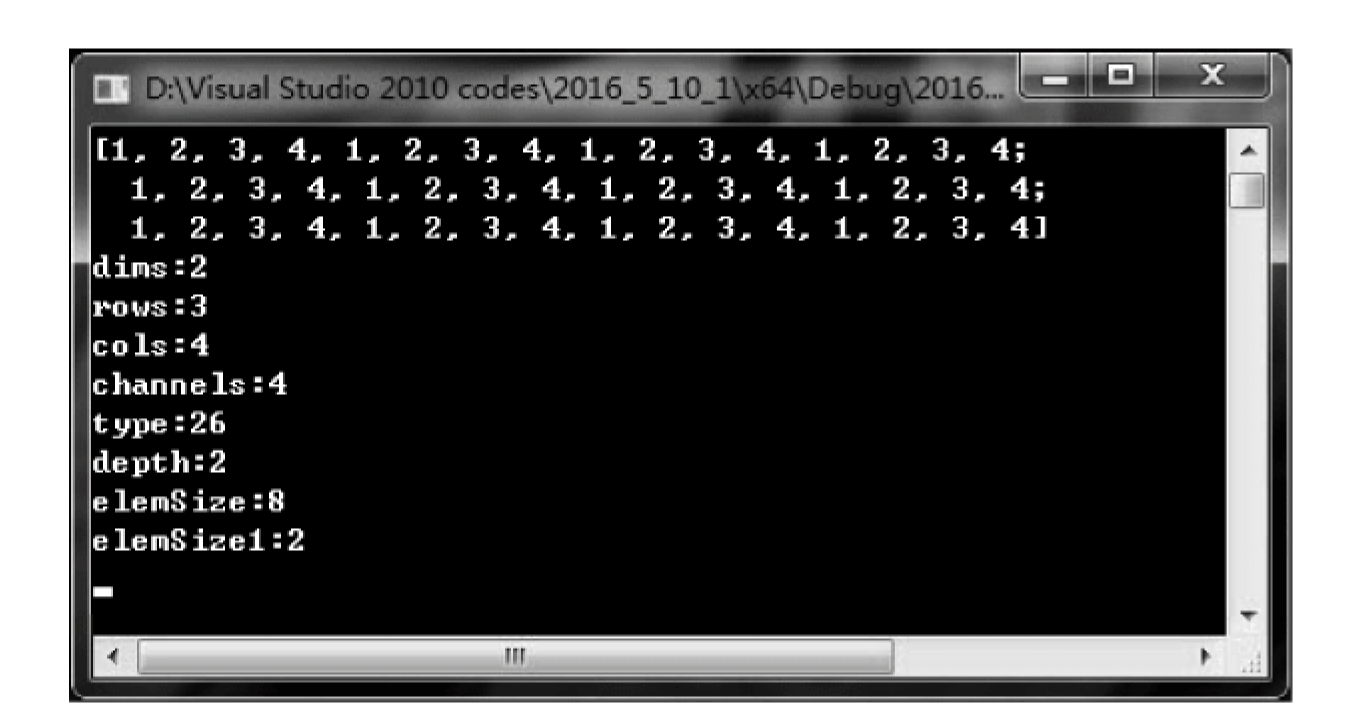


图 3-1　Mat 属性测试结果图

除以上介绍的几种 Mat 属性外，下面还要特别指出三种易混淆的 Mat 属性：step、step1 和 size。不过在此之前需要知道 OpenCV 内对于维度的定义。

OpenCV 对于维度的定义是：矩阵的深度为第一维，矩阵的高度为第二维，矩阵的宽度

为第三维。用一个例子来进行说明，如图 3-2 所示，该矩阵高度为 4，宽度为 5，深度为 3。矩阵的面就是其第一维，也就是矩阵深度，例子中该值为 3；矩阵中每个面上的每一行为第二维，也就是矩阵高度，例子中该值为 4；矩阵的每一行中的每一个点是第三维，也就是矩阵的宽度，例子中该值为 5，如图 3-3 所示。

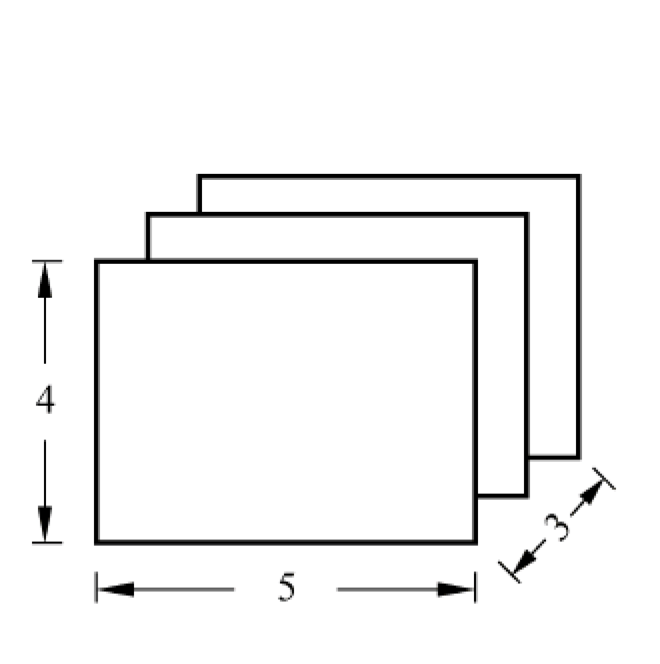


图 3-2　三维矩阵示意图

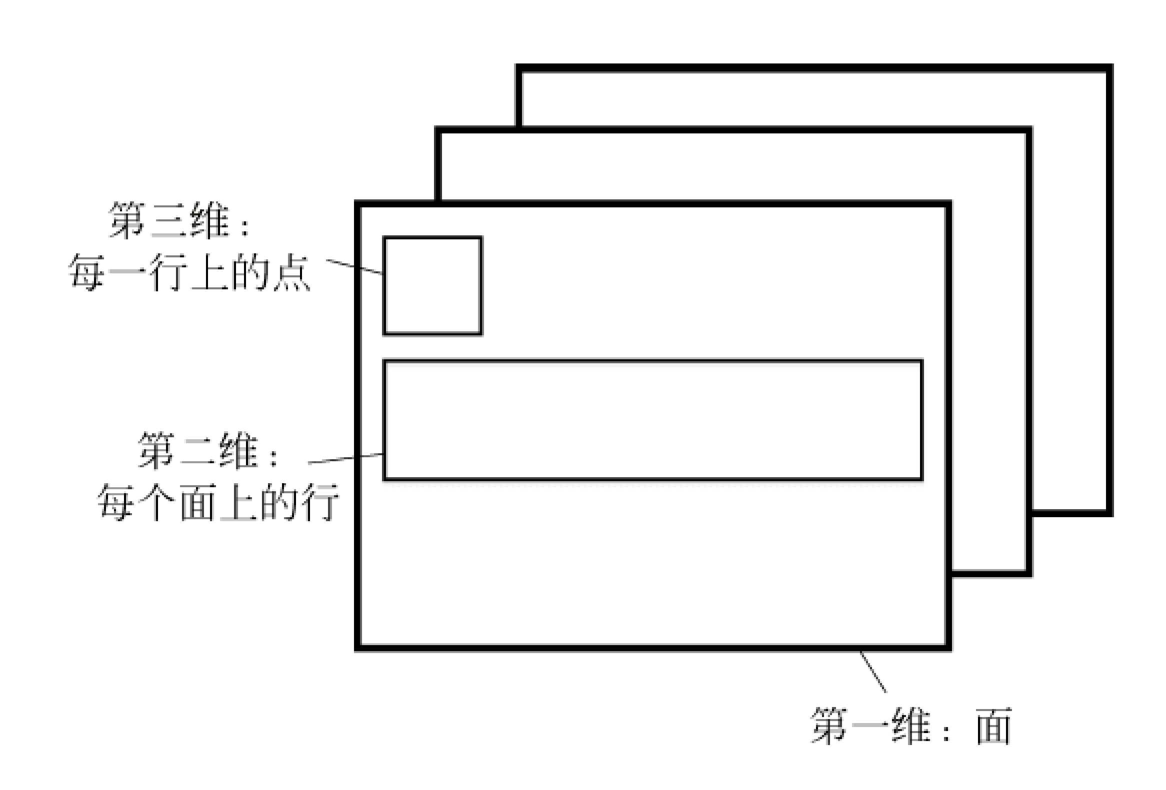


图 3-3　OpenCV 对三维矩阵的维度划分

（9）size。

size 表示每一维元素的个数，通常情况下会使用 size[0]、size[1]和 size[2]来分别表示矩阵的第一维、第二维和第三维。使用如图 3-2 所示的矩阵，对应的取值为 size[0]＝3，size[1]＝4，size[2]＝5。

（10）step

step 表示的是每一维的元素的大小，单位是字节。step 与 size 用法类似，也是用 step[0]、step[1]和 step[2]。step[0]表示第一维的元素的大小，即每个面的元素的总字节数；step[1]表示第二维的元素的大小，即面上的每一行的元素的总字节数；step[2]表示第三维元素的大小，即每一行中的一个元素的总字节数。

那么对于之前的矩阵，其 step 值为：

```
step[2] = 3 × 8 ÷ 8 = 3;
step[1] = step[2] × 5  =  3 × 8 ÷ 8 × 5  =  15;
step[0] = step[1] × 4  =  3 × 8 ÷ 8 × 5 × 4  =  60;
```

需要补充的是，如果矩阵是二维矩阵，那么 step[0]仍然表示一维元素的大小，但是这个时候，就不是面，而是单个面中的线，也就是一行元素所占的字节数量。同理，step[1]表示二维元素的大小，即矩阵中一个元素的大小，而 step[2]则没有意义。

（11）step1。

step1 表示每一维元素的通道总数，同样的 step1(0)是一维元素的通道总数，即每一个面上所有元素的通道数；step1(1)是二维元素的通道总数，即每一个面上的每一行的元素的通道数；step1(2)是三维元素的通道总数，即每一行上，每一个元素的通道数。还是用前面提到的三维矩阵来举例，那么有[4]：

```
step1(2) = 3;                //三通道
step1(1) = 3 × 5  =  15;
step1(0) = 3 × 5 × 4 =  60;
```

通过以下程序来解释一下 size、step 和 step1 的区别，程序如下：

```
#include<opencv2/opencv.hpp>
#include<iostream>
usingnamespace cv;
usingnamespace std;

void main()
{
    int matSize[] = {3,4,5};   //定义第一维为 3;第二维为 4;第三维为 5
    Mat img(3,matSize,CV_8UC3, Scalar::all(0));

    cout <<" ------ size 的大小 ------ "<< endl;
    cout <<"size[2] = "<< img.size[2] << endl;
    cout <<"size[1] = "<< img.size[1] << endl;
    cout <<"size[0] = "<< img.size[0] << endl;

    cout <<"\n ------ step 的大小 ------ "<< endl;
    cout <<"step[2] = "<< img.step[2] << endl;
    cout <<"step[1] = "<< img.step[1] << endl;
    cout <<"step[0] = "<< img.step[0] << endl;

    cout <<"\n ------ step1 的大小 ------ "<< endl;
    cout <<"step1(2) = "<< img.step1(2) << endl;
    cout <<"step1(1) = "<< img.step1(1) << endl;
    cout <<"step1(0) = "<< img.step1(0) << endl;

    getchar();
}
```

程序运行结果如图 3-4 所示。

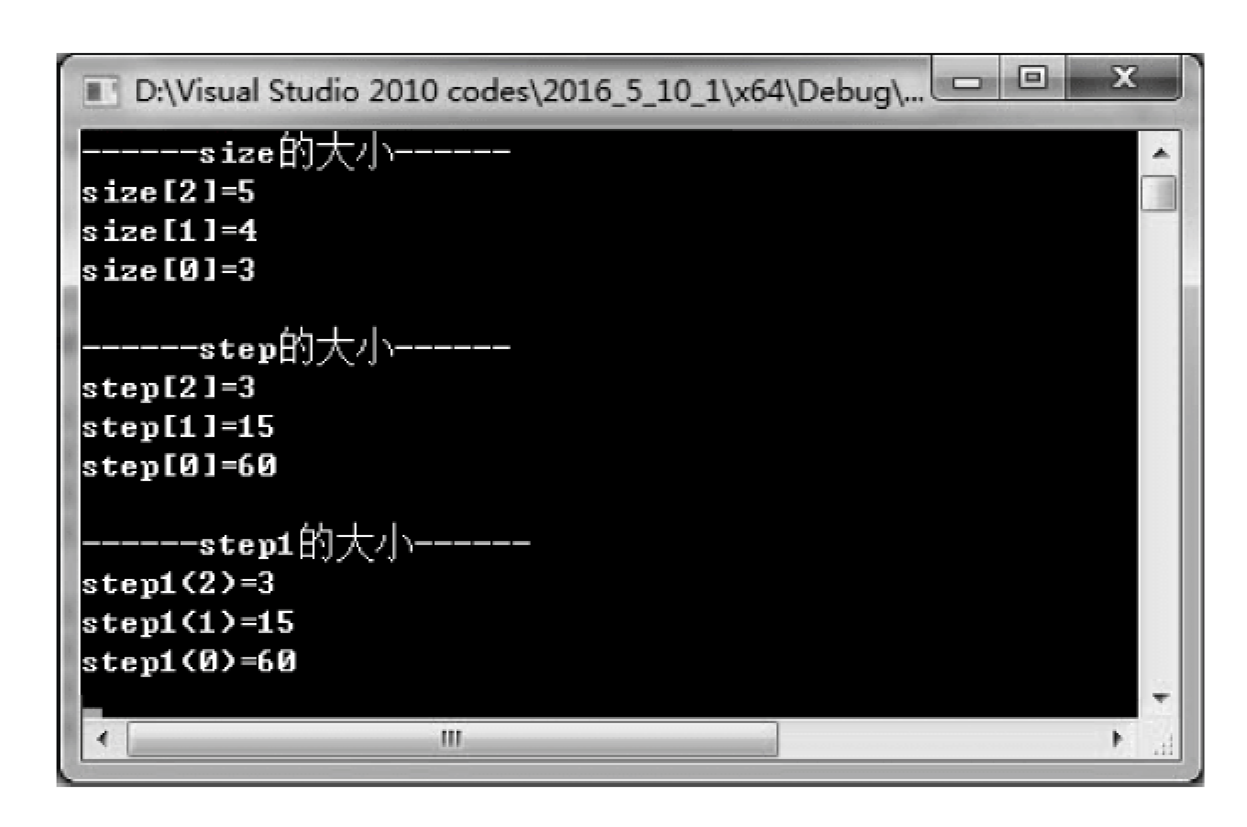


图 3-4　Mat 中 size、step 和 step1 的区别

3.1.2　imread 函数

imread 函数是 OpenCV 2 系列新推出的读入图像函数，其使用方式和 MATLAB 中的 imread 函数极为相似，使用方便，减少了编程时的难度。

imread 函数的定义为：

```
cv::Mat imread(const string & filename, int flag = 1);
```

(1) const string & filename。

这个参数是 const string 类型的 filename，需要在这里放入默认路径下的图像的名称，并且支持绝大多数的图像格式，如：JPEG 型中的.jpeg、.jpg 文件；Windows 位图型中的.bmp 文件；以及 PNG 格式下的.png 文件等。

(2) int flag=1。

这个参数是 int 类型的 flags，它作为一个图像读入的标识，指定了图像读入的颜色类型，也可以说成是指定读入通道的数量。若不指定 flag 参数，则其默认值为 1，代表读入三通道 RGB 彩色图像[5]。

其参数值有如下几种：

- flat=0——图像以单通道灰度图方式读入。
- flat=1——图像以三通道 RGB 方式读入。
- flat=2——图像深度若为 16 位或 32 位，则以相应的深度进行读入，如果不是深度为 16 位或 32 位，则会以 8 位图像读入。
- flat=其他——默认与 flat=1 对应的读入方式一致。在从前的版本中，flat=−1 表示读入 8 位有颜色图或灰度图，但是在新版本中已经被废置了，所以这里的负数部分无须太过深究。

imread()函数常用读取的例子如下：

```
Mat img = imread("Peashooter.jpg");                              //RGB
Mat img = imread("Peashooter.jpg",0);                            //灰度图
Mat img = imread("E:\VisualStudio2012_code\Peashooter.jpg");     //RGB
Mat img = imread("E:\VisualStudio2012_code\Peashooter.jpg",2);   //RGBA
```

3.1.3 imshow 函数

imshow()函数可以在窗口中显示图像，与 MATLAB 中 imshow()函数十分类似，其原型如下：

```
void cv::imshow( const std::string & winname, cv:: InputArray mat)
```

头文件：highgui.hpp，命名空间：cv。

(1) const std::string & winname

winname 是 const string& 类型的参数，为显示图像的窗口名称。

(2) cv::InputArray mat

mat 是 InputArray 参数，这个位置放入 Mat 类的变量，即图像存储的变量。

在使用 imshow 函数时需要注意图像显示过程中可能会进行缩放，其具体缩放的程度取决于原始图像的深度，具体如下：

① 图像是 8 位无符号型，则输出图像不会发生任何变化；

② 图像是 16 位无符号型或 32 位整型，会将像素值除以 255 显示出来。

③ 图像是 32 位浮点型，像素值需要乘以 255 显示出来。

结合 Mat 类、imread()和 imshow()来看一下读入一张三通道图像后 Mat 类内部变量值的情况，对应的程序如下：

```
#include <iostream>
#include <opencv2/opencv.hpp>
using namespace std;
using namespace cv;

int main(int argc, char * argv[])
{
    Mat img = imread("Peashooter.jpg");
    imshow("image", img);
     cout << "图像参数如下:" << endl;
     cout << "dims:" << img.dims << endl;
     cout << "rows:" << img.rows << endl;
     cout << "cols:" << img.cols << endl;
     cout << "channels:" << img.channels() << endl;
     cout << "type:" << img.type() << endl;
     cout << "depth:" << img.depth() << endl;
     cout << "elemSize:" << img.elemSize() << endl;
     cout << "elemSize1:" << img.elemSize1() << endl;

    cout << "\n------ size 的大小 ------ " << endl;
    cout << "size[2] = " << img.size[2] << endl;
    cout << "size[1] = " << img.size[1] << endl;
    cout << "size[0] = " << img.size[0] << endl;

    cout << "\n------ step 的大小 ------ " << endl;
    cout << "step[2] = " << img.step[2] << endl;
    cout << "step[1] = " << img.step[1] << endl;
    cout << "step[0] = " << img.step[0] << endl;

    cout << "\n------ step1 的大小 ------ " << endl;
    cout << "step(2) = " << img.step1(2) << endl;
    cout << "step(1) = " << img.step1(1) << endl;
    cout << "step(0) = " << img.step1(0) << endl;

     waitKey(0);              //按任意键退出
     return 0;
}
```

程序运行结果如图 3-5 和图 3-6 所示。

以上程序需特别注意一点：图像是二维的，因此 size[2]、step[2]和 step1[2]会出现乱码，这三项没有参考价值。

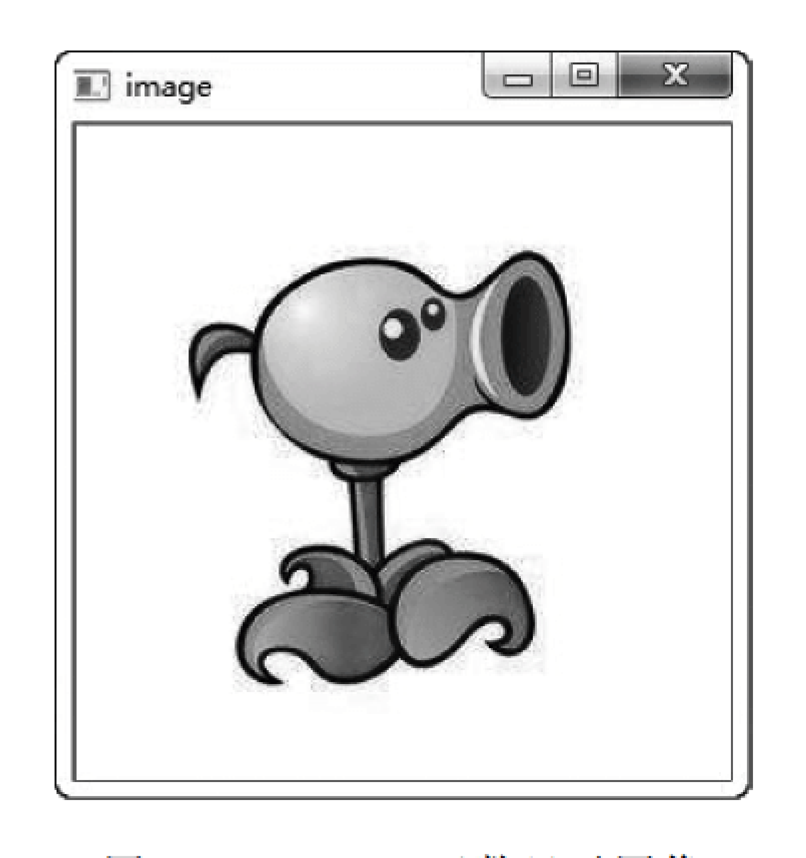


图 3-5　imshow 函数显示图像

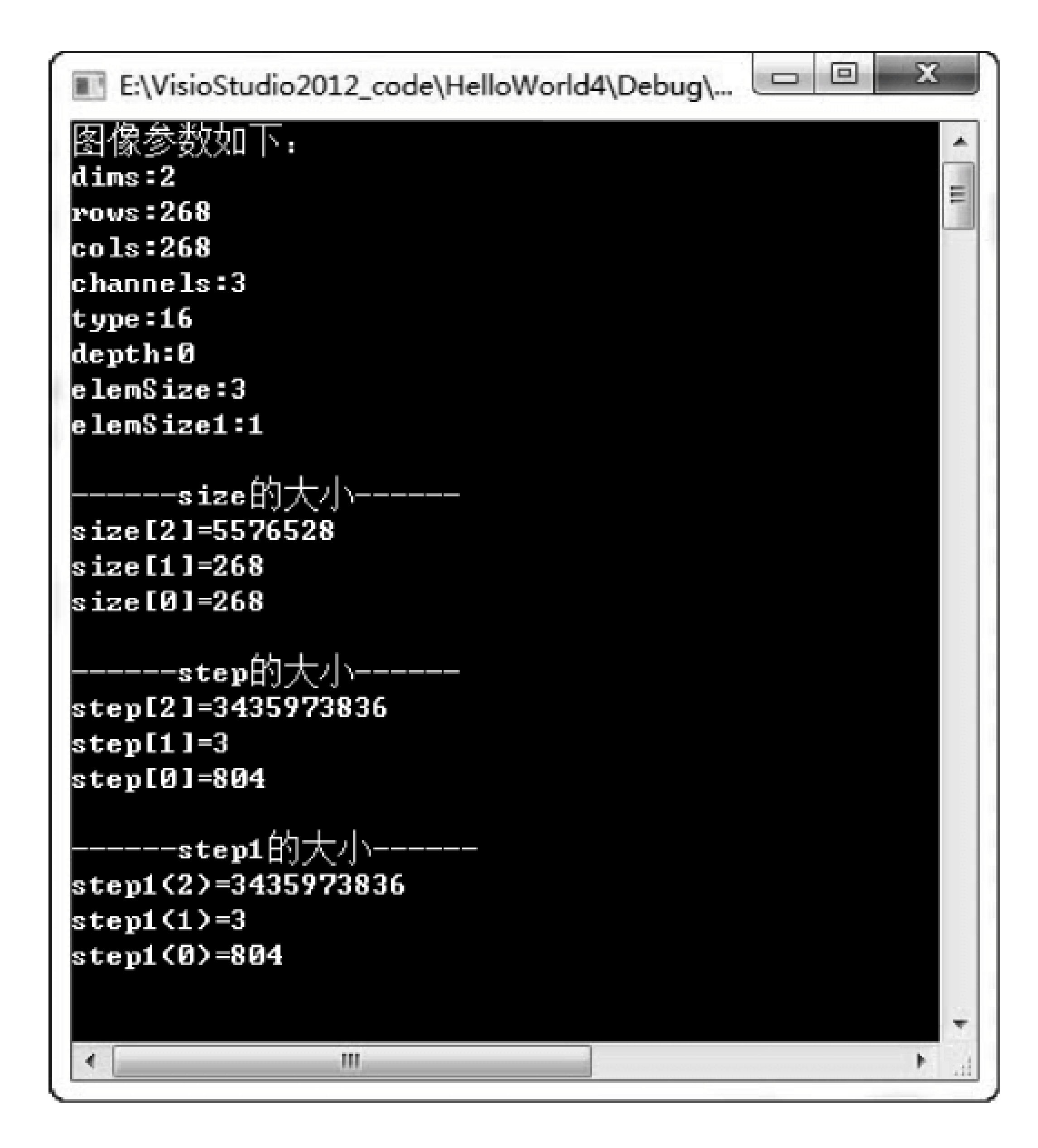


图 3-6　使用 imshow 得到图像参数

3.1.4　imwrite 函数

imwrite()函数通常用于存储处理后的图像，函数声明如下：

```
bool cv::imwrite(const std::string &filename, cv::InputArray img,const std::vector < int,
std::allocator < int >> &params  =  std::vector < int >())
```

头文件：highgui.hpp，命名空间：cv。

（1）const std::string &filename。

filename 是 const string& 类型的，表示文件名及存储格式。

（2）cv::InputArray img。

img 是准备被保存起来图像的 Mat 类参数。

（3）const std::vector < int，std::allocator < int >> &params=std::vector < int >()。

params 是 const vector < int >类型的参数，表示特定格式残存的参数编码。其默认值为 vector < int >()。

3.2　反向算法

1. 反向算法的原理

反向算法是图像处理中最简单的算法之一，适合用于环境搭建测试，反向算法即将图像中所有的像素点的色彩度反色，算法原理如下面公式所示：

$$f(x) = 255 - g(x) \tag{3-1}$$

因书中图是黑白图，很难看到颜色变化，如图 3-7 所示为预期的效果图。

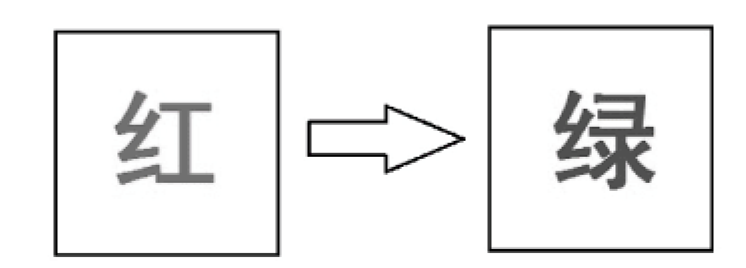


图 3-7　反向算法示意图

2. 反向算法实现

```
//------------------------------- 头文件部分 -------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;


//------------------------------ 主函数部分 --------------------------------
int main(int argc, char * * argv)
{
    IplImage * srcImg = cvLoadImage("lena512.png",1);
//读入图像 lena512.png

//------------------------- 计时函数:开始计时段 ----------------------------
    double timeSpent = (double)getTickCount();


//------------------------------ 提取图像信息 ------------------------------
    Int iii,jjj;
    int nHeight,nWidth,nChannels,nWidthStep;
    nHeight = srcImg->height;                    //使用 OpenCV 提取图像高度
    nWidth = srcImg->width;                      //使用 OpenCV 提取图像宽度
    nChannels = srcImg->nChannels;               //使用 OpenCV 提取图像通道数
    nWidthStep = srcImg->widthStep;              //使用 OpenCV 截取一行像素点宽度

//------------------------------ 算法实现部分 ------------------------------
uchar * data = (uchar * )srcImg->imageData;
        //三通道彩色图像,使用两套循环遍历所有像素点实现反向
    for(jjj = 0;jjj < nHeight;jjj++)
    {
        for(iii = 0;iii < nWidth * nChannels;iii++)
        {
    data[jjj * nWidthStep + iii] = 255 - srcImg->imageData[jjj * nWidthStep + iii];
        }
    }

//------------------------------ 计时函数:终止阶段 --------------------------
```

```
    timeSpent = ((double)getTickCount() - timeSpent)/getTickFrequency();
    cout <<"Time spent in milliseconds: "<< timeSpent * 1000 << endl;
//在屏幕上输出使用的时间

    cvShowImage("反向图像",srcImg);              //显示图像
    cout <<" Channels "<< nChannels << endl;    //显示图像通道数
    waitKey(0);                                 //循环等待退出
    return 0;
}
```

3. 程序运行结果

程序运行结果如图 3-8 和图 3-9 所示，左图为原始图像，右侧为处理后的图像。



图 3-8 反向算法的结果 1

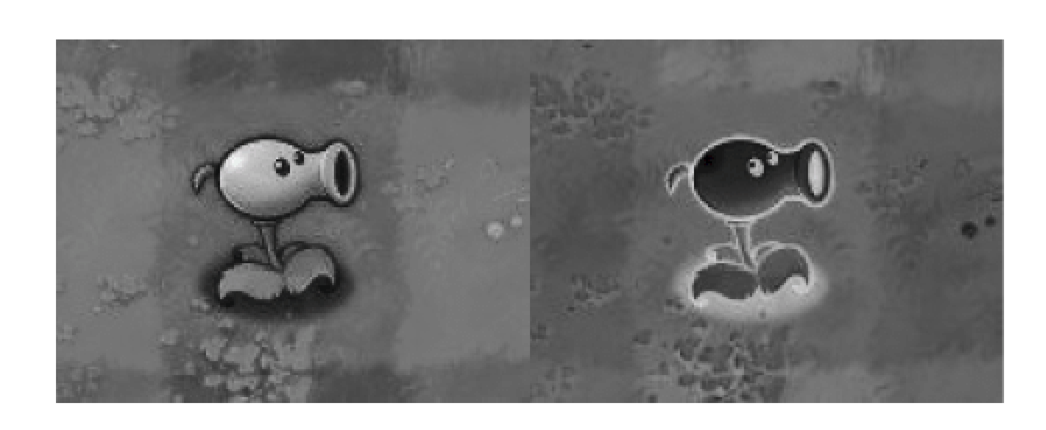


图 3-9 反向算法的结果 2

4. 程序简析

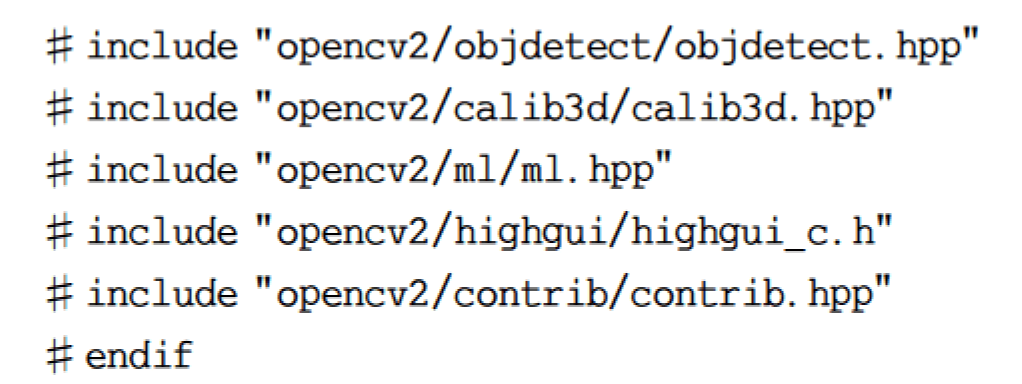
反向算法相对浅显，相信读者根据程序中的备注，可以看懂该算法。因此比较容易理解以反向算法为例介绍 OpenCV 编程时会涉及的常识以及需要注意的细节。

1）包含头文件部分

编写 OpenCV 程序时，经常会使用名为 opencv.hpp 的头文件，该头文件中有如下定义[6]：

```
#inlude <opencv2/opencv.hpp>
#ifndef __OPENCV_ALL_HPP__
#define __OPENCV_ALL_HPP__

#include "opencv2/core/core_c.h"
#include "opencv2/core/core.hpp"
#include "opencv2/flann/miniflann.hpp"
#include "opencv2/imgproc/imgproc_c.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/photo/photo.hpp"
#include "opencv2/video/video.hpp"
#include "opencv2/features2d/features2d.hpp"
```



```
#include "opencv2/objdetect/objdetect.hpp"
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/ml/ml.hpp"
#include "opencv2/highgui/highgui_c.h"
#include "opencv2/contrib/contrib.hpp"
#endif
```

通过以上定义可以知道，opencv.hpp 头文件包含了绝大多数常见的 OpenCV 头文件。因此，编程时仅写#inlude <opencv2/opencv.hpp>，在实现功能的同时，也可以实现程序的精简化。

2）头文件命名部分

```
using namespace cv;
using namespace std;
```

这两行程序说的是文件的命名空间在 cv 之内，通常对文件进行命名有两种方法：

第一种方法如前面程序所写，程序最开头，包含头文件的部分直接添加上 using namespace cv，这样后续所有的类均在 cv 之内。

第二种方法是在程序开头部分不加 cv 声明，但后面每一个 cv 类或函数，都要以“cv::”这种方式命名，这种命名方式虽然正确但是过于烦琐，且易出错，因此不推荐。

- 主函数部分：main()函数的写法。

例程中的 main()函数与 C 语言和 C++语言的 main()函数命名稍有不同，这里写法为：

```
int main(int argc, char * * argv)
```

其中 arg 指的是参数，argc 为整数，用来统计运行程序时送给 main()函数的命令行参数的个数。argv 同上会加上 * 和[ ]，成为 * argv[ ]，表示字符串数组，用来存放指向字符串参数的指针数组，每一个元素指向一个参数。

main()函数在使用过程中，不论写不写上 argc 和 argv 都是正确的，因此 main()函数通常有如下两种写法：

① int main(int argc, char * * argv) { }

② void main() { }

- 图像的读入：IplImage * srcImg = cvLoadImage("lena512.png",1)。

示例采用 opencv 1 系列中图像读入的方法，IplImage 函数使用了 C 语言作为接口进行读入。在 OpenCV 2.4.X 系列和之后的版本中，基本已经舍弃了这种读入方式，取而代之的是用 Mat 类进行读入。但本算法需要使用指针，所以没有把图像变成 Mat 类。

这两种读入方式用法如下：

```
IplImage * srcImg = cvLoadImage("lena512.png",1);
Mat srcImg = imread("lena512.png",1);
```

前者为 C 接口的指针型，后者为 C++接口的 Mat 类。

① 读入方式。

以上两种读入方式都支持的图像格式有：

```
Windows 位图：.bmp、.dib
```

```
JPEG 文件：.jpeg、.jpg、.jpe
JPEG2000 文件：.jp2
PNG 文件：.png
便携文件：.pbm、.pgm、.ppm
Sun rasters 光栅文件：.sr、.ras
TIFF 文件：.tiff、.tif
```

② 载入标识 flags。

载入标识指在图像读入过程中对有颜色图像加载的颜色类型，flags 不同对应读入计算机中的图像颜色也会有区别，通常不输入 flags 时，其默认值为 1。与前面介绍的 Mat 类的 flags 标识含义完全相同。

- 计时函数：getTickCount()。

计时开始函数：

```
double timeSpent = (double)getTickCount();
```

计时终止函数：

```
timeSpent = ((double)getTickCount() - timeSpent)/getTickFrequency();
```

getTickCount 函数的作用是返回(retrieve)从操作系统启动所经过(elapsed)的毫秒数(ms)，它的返回值是 DWORD。

使用方法如上所述，只要设定好一个变量，按照上面的格式去写就可以测试这两个变量中间的程序运行所消耗的时间[7]。

- 提取图像信息。

```
nHeight = srcImg -> height;
nWidth = srcImg -> width;
nChannels = srcImg -> nChannels;
nWidthStep = srcImg -> widthStep;
```

本算法使用 IplImage 读取图像的信息，其中 nHeight、nWidth、nChannels、nWidthStep 分别指存储图像的高度 height、图像的宽度 width、图像的通道数 nChannels 和每一行像素点存储所需要的字节数 nWidthStep，如图 3-10 所示。其中 height、width、nChannels 和 widthStep 是 OpenCV 自带的函数，专门用来进行图像的特征提取。

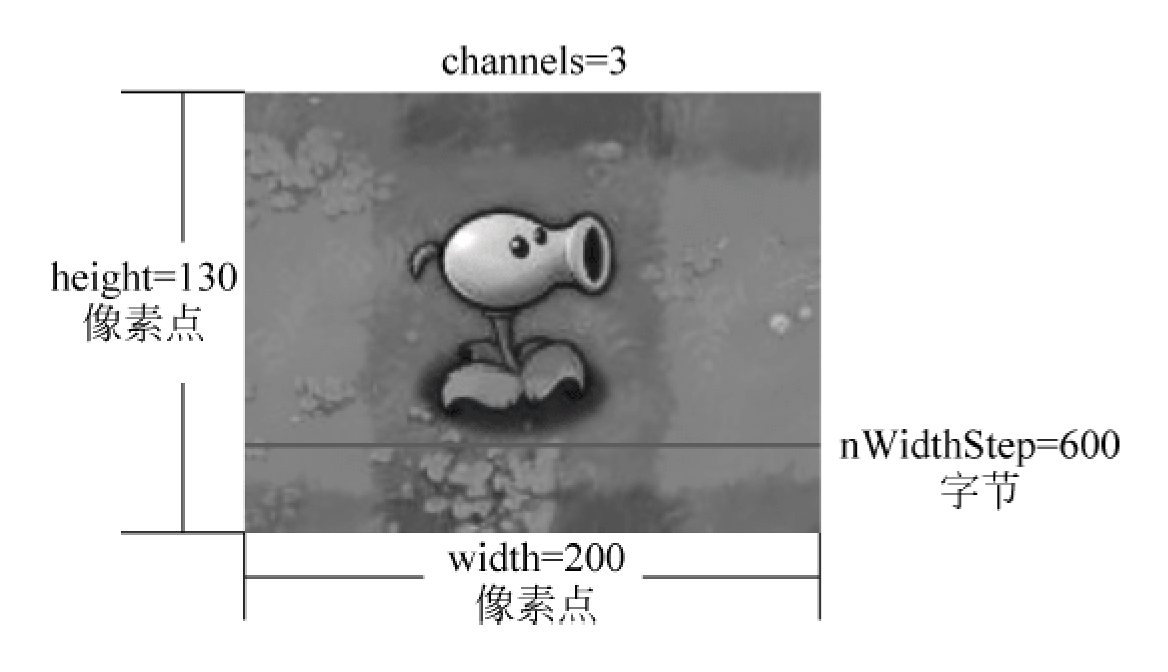


图 3-10 提取图像信息的变量

需要注意的是，nWidthStep 指存储一行图像的像素点所需要的字节数，其自身必须为 4 的倍数，以便实现字节对齐，这种方式会提高运算速度。一般情况下 nWidthStep＝width×nChannels，即每一行所需要的字节数等于每一行像素点的数量乘图像自身的通道数，但是如果图像自身没有令 nWidthStep 为 4 的倍数，那么这个计算方式就不成立，需要使用空字节来补齐。

使用上面的图像进行说明：

上述存储在计算机中的三通道的彩色图，其数据存储效果如图 3-11 所示，以 B、G、R 的形式进行存储。像素点 width＝200，通道数 nChannels＝3，所以刚好可以实现 nWidthStep＝600 是 4 的倍数。

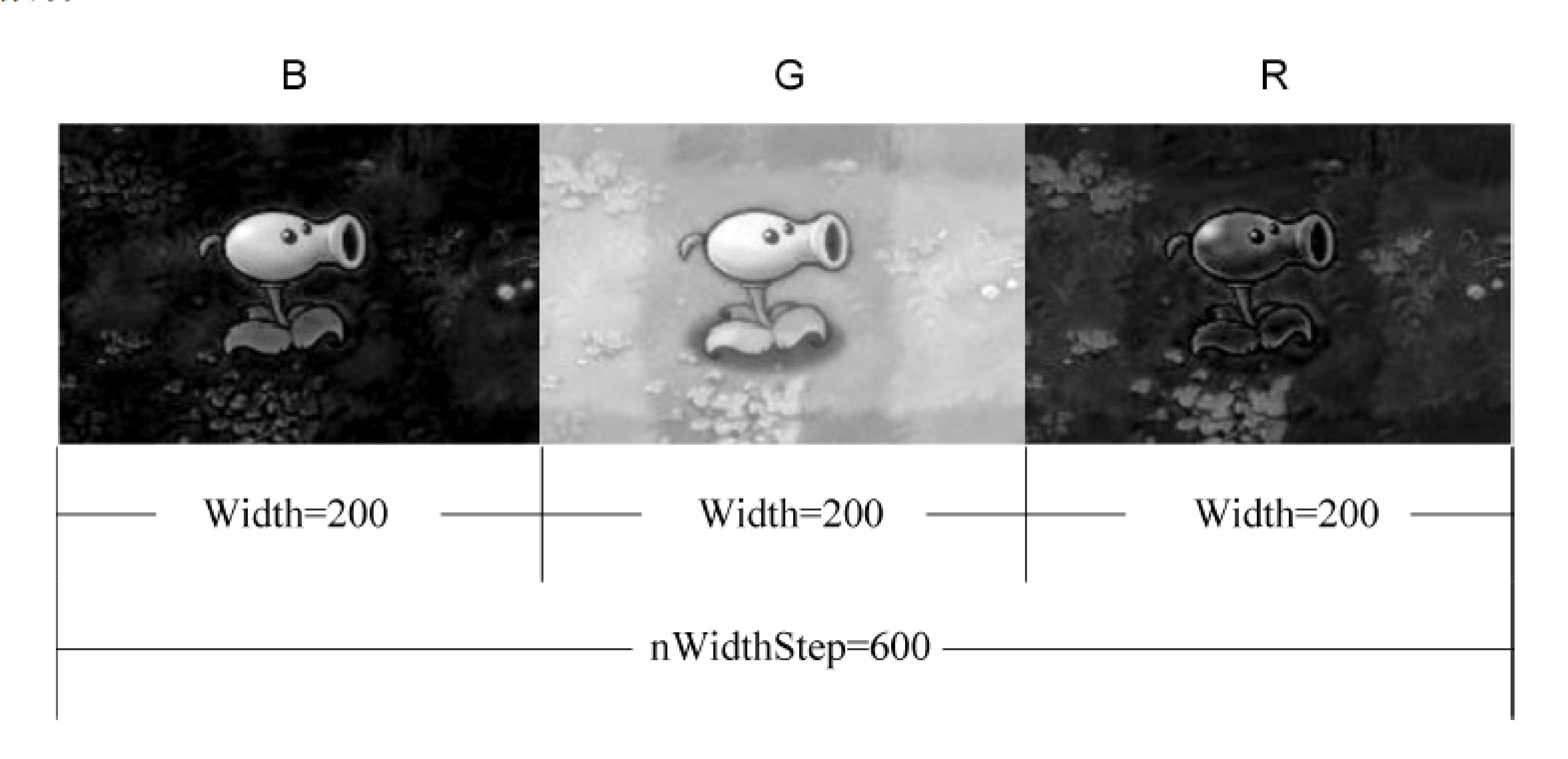


图 3-11　计算机中存储三通道图像

假设一行像素点为 199 个，即 width＝199，nChannels＝3，理论上存储在计算机中的 nWidthStep 应该是 199×3＝597，但是在实际应用的过程中 nWidthStep 的值会是 600，因为 597 不是 4 的倍数，遵照向上对齐的原则使读入的图像后边添加了三个空的字符，这三个字符仅仅是用来实现字符对齐的，没有其他任何作用，如图 3-12 所示。

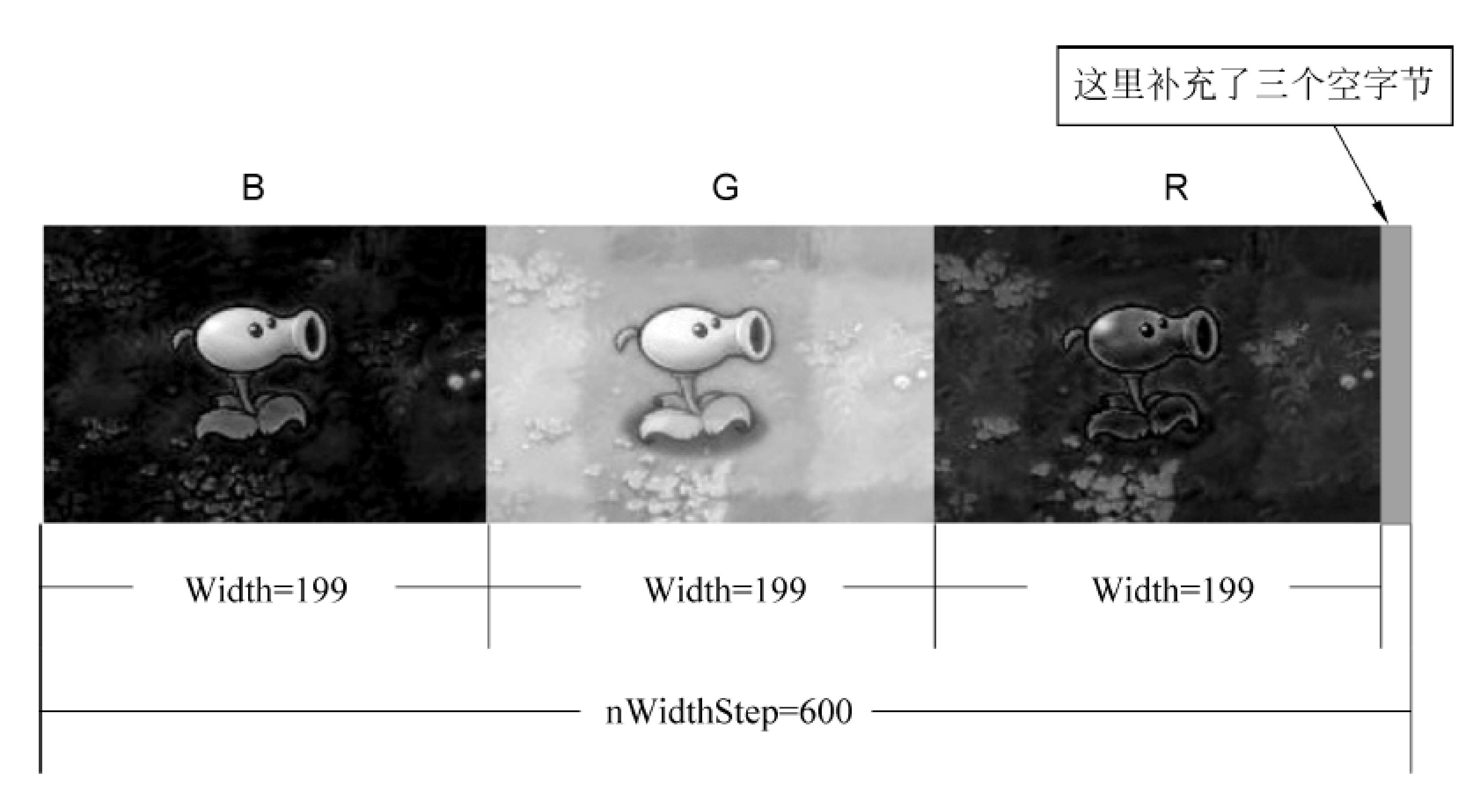


图 3-12　计算机中 nWidthStep 存储补足

这个对齐的模式可能会对一些特殊的图像处理算法造成影响，例如 NLM 算法在外围搜索的过程中会搜索到边缘没有意义的像素点。所以建议在处理图像时尽量选择标准大小的图像，如像素点为 64×64、128×128、256×256……或者图像的一行像素点为 4 的倍数也可以。

• 算法部分：

```
for(jjj = 0;jjj < nHeight;jjj++)
{
    for(iii = 0;iii < nWidth * nChannels;iii++)
    {
data[jjj * nWidthStep + iii] = 255 - srcImg -> imageData[jjj * nWidthStep + iii];
    }
}
```

程序使用了两套循环来实现图像的遍历。图 3-13 为三通道彩色图像存入计算机中显示效果图，OpenCV 和 MATLAB 显示的部分都一样，都是三通道三个矩阵的形式进行显示，图下方的方框示意为图像中像素点的具体数值。

B　G　R

12	67	78	67
51	77	87	57
52	76	7	51
125	87	76	12

32	76	56	97
46	52	144	86
12	26	89	12
66	80	51	44

53	99	100	49
57	77	56	42
54	77	90	45
166	97	209	72

图 3-13　计算机中存储三通道图像示意图

在计算机存储的过程中，OpenCV 和 MATLAB 的存储方式有一些区别，MATLAB 会将单通道灰度图像存成二维平面矩阵，将三通道或者多通道彩色图像存成三维空间矩阵，例如现在想取 Green 层第二行第二列的像素点 46，MATLAB 就可以写成：

```
A = Data[1,1,1]
% A 为变量,Data 为图像
%第一个 1 是指第二行,第二个 1 指第二列,第三个 1 为第二层 Green 层
```

OpenCV 的存储方式是将所有的 BGR 三色像素点以一个一维数组的方式进行存储的，如图 3-14 所示，第一行存储完成之后直接存储第二行，这也就是为什么 OpenCV 要用到 nWidthStep 函数。如果图像 width 不是 4 的倍数，导致 nWidthStep 需要补充时，补充的空白字节也将以这种方式存在这个一维数组中。

了解 OpenCV 的存储方式后，之前的遍历算法也就清楚了：

jjj(行)×nWidthStep(每一行所需的字节数)+iii(列)=像素点位置

• 图像显示：cvShowImage("反向图像",srcImg)。

图形显示也有两种方式：一种是例子中使用的 cvShowImage()函数，这个函数可以将指针形式存储的图像无损地显示出来。第一个参数是“窗口名称”，用双引号(一定注意是英文输入法下的双引号)将名称包括在中间即可；第二个参数是图像的指针名称。

12	67	78	67	32	76	56	97	53	99	100	49
51	77	87	57	46	52	144	86	57	77	56	42
52	76	7	51	12	26	89	12	54	77	90	45
125	87	76	12	66	80	51	44	166	97	209	72

Data=[12,67,78,67,32,76,56,97,53,99,100,49,51,77,87,57,46,...]

图 3-14 OpenCV 存储像素点的模式

第二种输出函数较为常用：imshow()函数。

imshow()函数可以输出 Mat 类读入的图像，使用起来也更方便，因此这种方式更为常用，使用方式如下：

```
imshow("图像显示",image);
```

第一个参数是在双引号内写出图像显示窗口的名称，后一个参数是用来存储图像的 Mat 类变量。

5. 附加

为了对比 OpenCV 函数和 MATLAB 函数的区别，这里使用 MATLAB 进行一次反向算法编写，程序如下：

```
clear all
f = imread('PeaShooter.png');              % 图像读入
tic                                        % MATLAB 的计时函数
[M,N,Z] = size(f);                         % 图像高度、宽度、通道数测量
g = ones(M,N,Z);                           % 建立一个全是 1 的矩阵
for i  =  1:M                              开始进行像素点遍历
for j = 1:N
for k = 1:Z
            g(i,j,k)  =  255  -  f(i,j,k); % 实现反向
end
end
end
g = uint8(g);                              % 将浮点型数据转化为整型
figure;                                    % 画图
subplot(121);                              % 在图中左侧放上原图像,右侧放上反向后的图像
imshow(f,[]);
subplot(122);
imshow(g,[]);
toc
```

为了方便对比，这次实现仍使用了与之前一样的图像，处理的结果如图 3-15 所示，与使用 OpenCV 的处理结果一模一样，但是处理的时间却远超过 OpenCV。

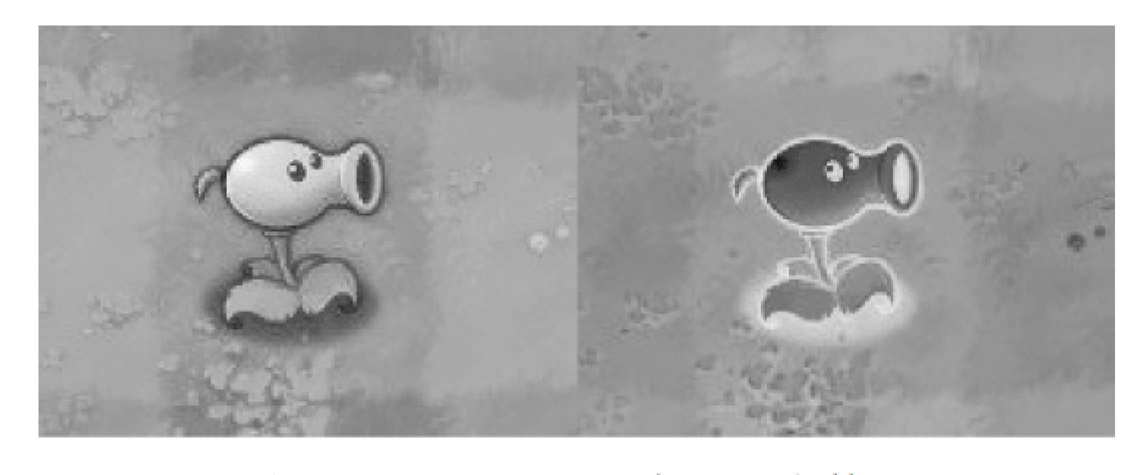


图 3-15 MATLAB 实现反向算法

3.3　图像融合

图像融合是图像处理中比较常用的一种处理手段，将两个或多个图像以一定的方式融合成一个图像。如图 3-16 所示，左侧为静物的左聚焦图像，图中间为静物的右聚焦图像，图右侧为两个图像中清晰部分融合起来的图像。



图 3-16　图像融合示例

本节将使用 OpenCV 自带函数实现简单的图像融合或者叫图像混合，通过这些例程可以加深对 OpenCV 函数的理解。

3.3.1　覆盖型图像融合

这种图像融合是将一个图像以马赛克的形式盖在另外一个图像的某个部位上。

（1）程序如下：

```
#include<opencv2/opencv.hpp>
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
using namespace cv;
using namespace std;

int main()
{
    Mat srcImage1 = imread("1.jpg");            //读取图像 1 作为背景
    Mat srcImage2 = imread("2.jpg");            //读取图像 2 作为覆盖面
    Mat imageROI = srcImage1(Rect(100,100,srcImage2.cols,srcImage2.rows));
    //ROI 部分
    Mat mask = imread("2.jpg",0);               //读入图像
    srcImage2.copyTo(imageROI,mask);            //图像覆盖
    namedWindow("图像融合");                    //给窗口命名
    imshow("图像融合",srcImage1);               //图像输出
    waitKey();
    return 0;
}
```

（2）实验结果。

相信通过程序备注，读者可以理解程序的含义，程序运行的结果如图 3-17～图 3-19 所示，图 3-17 为图像的背景，图 3-18 为背景上添加的图像，图 3-19 为完成示意图。



图 3-17　背景



图 3-18　背景上的图像

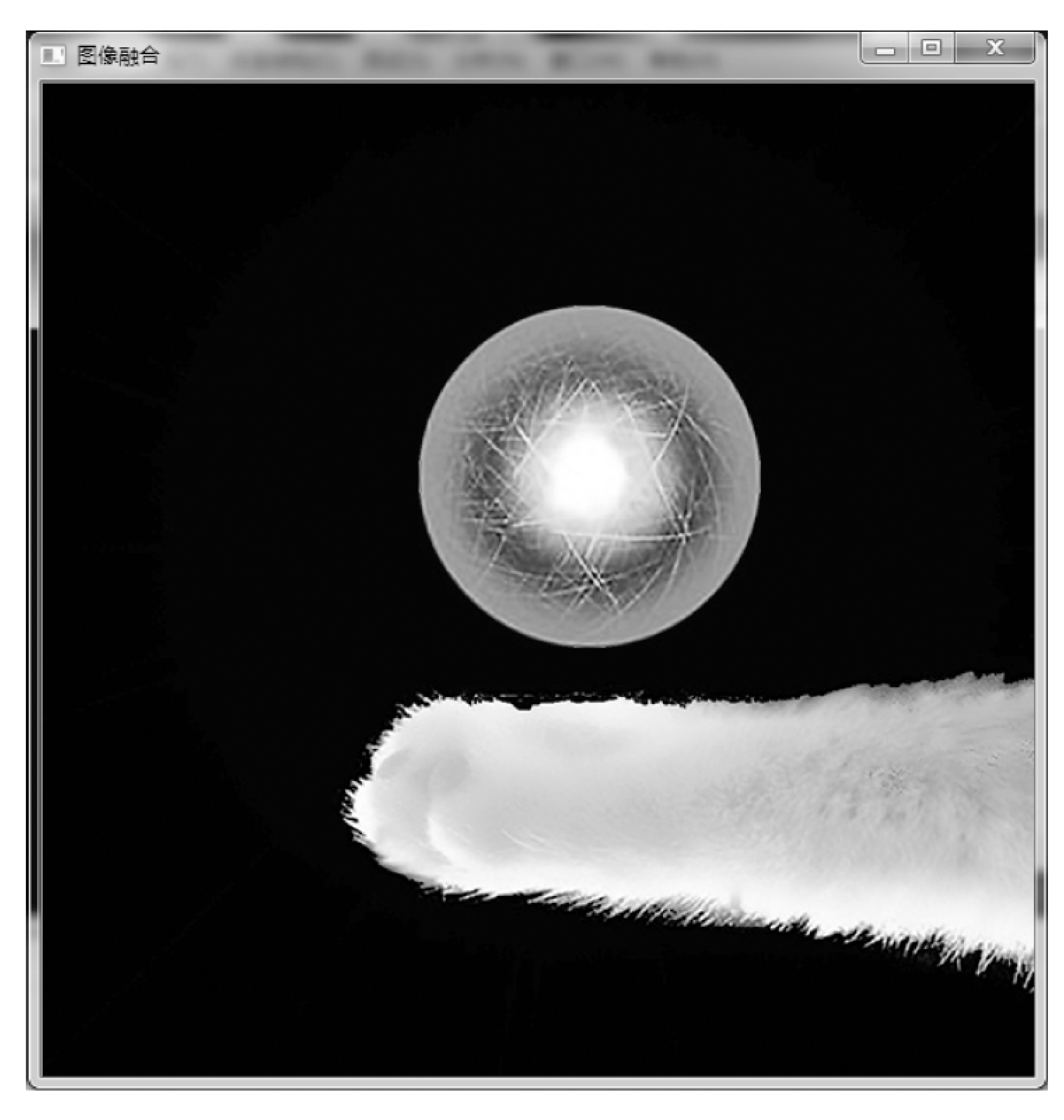


图 3-19　两幅图像融合后的结果

(3) 注意事项和程序简析。

ROI(Region Of Interest)指图像中感兴趣的部分,其实就是找到图像中想要进行图像处理的部分,通常使用 Rect 函数和 Range 函数,其中 Rect 函数更为常用,Rect 函数的使用方法如下:

```
X = A(Rect(B,C,D,E));
```

X 是一个 Mat 类的变量,用来存放图像中的 ROI;

A 是原图像;

B、C 用来确定 ROI 左上角点的坐标,其中 B 是原图像 A 中的第多少列,C 是原图像 A 中的第多少行;

D 和 E 是 ROI 的高度和宽度,D 是从 B、C 确定的原点位置,向下 D 个像素点,作为 ROI 的高度;E 是从 B、C 确定的原点位置,向右 E 个像素点,作为 ROI 的宽度。如图 3-20 所示。

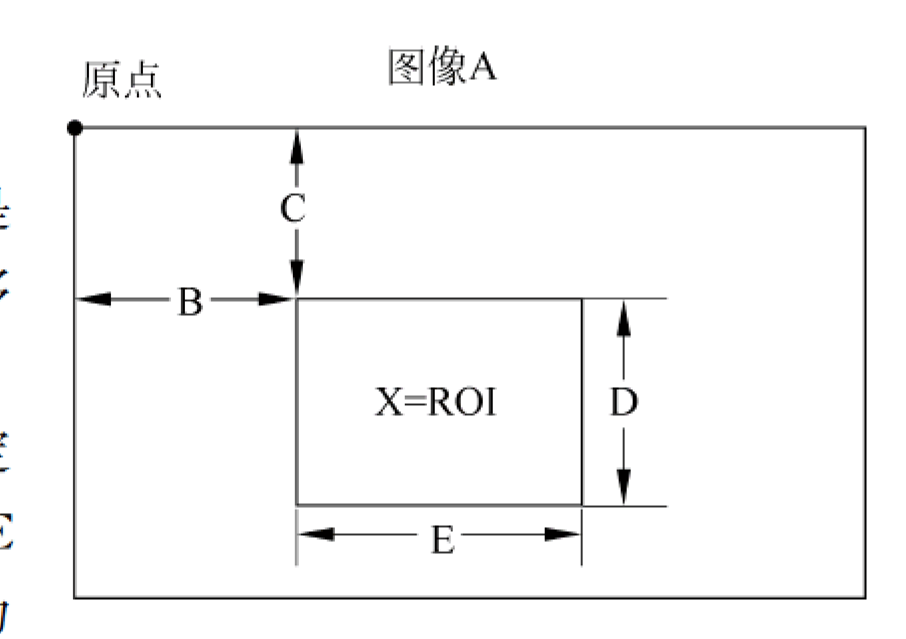


图 3-20　ROI 示意图

例如,在上述的程序中:

```
Mat imageROI = srcImage1(Rect(100,100,srcImage2.cols,srcImage2.rows));
```

imageROI 是原图像的 ROI 部分;

srcImage1 是原图像;

100,100 是 ROI 左上角的原点坐标。

srcImage2.cols 和 srcImage2.rows 是从 srcImage2 图像中获取的该图像的高度和宽度,其中.cols 是图像的高度而.rows 是图像的宽度。需要注意的是,ROI 要和掩膜(mask)的大小一致。

3.3.2　线性图像混合

线性图像混合是将图像中的像素点进行叠加,应用的是 addweight()函数,其公式如下:

$$g(x) = (1 - a)f_1(x) + af_2(x) \tag{3-2}$$

将两幅像素点相同、尺寸相同的图像,根据权重叠加起来即为线性图像混合,通过这种权重叠加,不会出现像素值超出 255 的情况。

(1) 程序如下:

```
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
using namespace cv;

void main()
{

double a =  0.6;                            //预设值
double b = 1 - a;
     Mat srcImage1, srcImage2, dstImage;

     srcImage1 =  imread("1.jpg");           //读取图像 1
     srcImage2 =  imread("2.jpg");           //读取图像 2

     namedWindow("原图像 1");                //显示图像 1
     imshow("原图像 1", srcImage1 );
     namedWindow("原图像 2");                //显示图像 2
```

```
    imshow("原图像 2", srcImage2 );

    addWeighted(srcImage1, a, srcImage2, b, 0, dstImage);
                                          //混合图像 1 和图像 2 并存于 dstImage 中

    namedWindow("线性混合结果");            //输出混合图像 dstImage
    imshow("线性混合结果", dstImage );

    waitKey();

}
```

(2) 处理结果。

图 3-21 和图 3-22 为需要混合的两幅图像,图 3-23 为二者线性混合后的图像。



图 3-21　线性混合图像原图 1



图 3-22　线性混合图像原图 2



图 3-23　线性混合结果图

(3) 程序简析。

本程序主要使用了 addWeight()函数,函数声明如下:

```
void addWeighted(InputArray src1, double alpha, InputArray src2, double beta, double gamma,
OutputArray dst, int dtype = -1);
```

- src1

第一个参数是 InputArray 类型的 src1,存放第一个图像,即要用来加权的第一个数组。

- alpha

第二个参数是 double 类型的 alpha,存放第一个加权数组的权重。

- src2

第三个参数是 InputArray 类型的 src2,存放第二个图像,即要用来加权的第二个数组。

- beta

第四个参数是 double 类型的 beta,存放第二个加权数组的权重。

- gamma

第五个参数是 double 类型的 gamma,存放另外一个加权到该权重上的标量值,可以用来调整图像整体的颜色深度等。

- dst

第六个参数是 OutputArray 类型的 dst,存放即将输出的数组,即作为输出的图像,该图像与原图像的大小和属性必须相同。经过 addWeighted()函数的处理,输出的 dst 像素点的公式如下:

$$dst = src1 \times alpha + src2 \times beta + gamma$$

- dtype

第七个参数是 int 类型的 dtype,存放输出数组 dst 的深度,其默认值为-1,表示与原本的输入数组 src1 和 src2 相同的图像深度。

(4) 注意事项。

addweight()函数仅仅适用于三通道和单通道的图像,如果是 16 位四通道或者是 32 位五通道的图像就不能使用这种方式去实现。

3.3.3 动画效果的线性混合

图像处理偶尔会出现需要图像动态显示的过程,本节仍然使用前面的线性混合函数 addWeighted(),并给出一个如何做简单动态效果的例程——将彩色图像渐变成灰度图像。

(1) 程序如下:

```
#include"stdafx.h"
#include<opencv2/core/core.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<iostream>

using namespace cv;
using namespace std;

int main(int argc, char **argv)
{
    Mat srcImg = imread("Jayce.jpg");
    namedWindow("原图像");
    imshow("原图像", srcImg);

/*----------------------------计时函数终止端---------------------------*/
    double timeSpent = (double)getTickCount();
/*----------------------------------------------------------------------*/
```

```
    Mat tmpImg1;
    Mat tmpImg2;

    cvtColor(srcImg,tmpImg1,CV_RGB2GRAY);
//将彩色的原图像 srcImg 转换成单通道的灰度图像,并存储在 tmpImg1 中

    cvtColor(tmpImg1,tmpImg2,CV_GRAY2RGB);
//将单通道的灰度图像 tmpImg1 转换成三通道的彩色图像,并存储在 tmpImg2 中

    Mat dstImg;                                          //定义输出图像
    double a;                                            //定义权重
    for(int i = 0; i < 100; i++)
    {
        a = (double)i/100.0;
        addWeighted(srcImg,1 - a,tmpImg2,a,0,dstImg);

        //addWeighted(srcImg,a,tmp3Img,1 - a,0,dstImg);   //反过来的灰度变彩色的过程
        namedWindow("渐变图像");
        imshow("渐变图像",dstImg);
        waitKey(20);                                     //控制渐变速度
    }

/* ---------------------------- 计时函数终止端 ---------------------------- */
    timeSpent = ((double)getTickCount() - timeSpent)/getTickFrequency();
    cout <<"Time spent in milliseconds: "<< timeSpent * 1000 << endl;
/* ---------------------------------------------------------------------- */

    waitKey(0);
    return 0;
}
```

(2) 运行结果。

如图 3-24 所示为原始图像,之后会随着时间渐渐变为如图 3-25 所示的图像,最后渐变为如图 3-26 所示的灰度图像。



图 3-24　原始图像



图 3-25　渐变图像

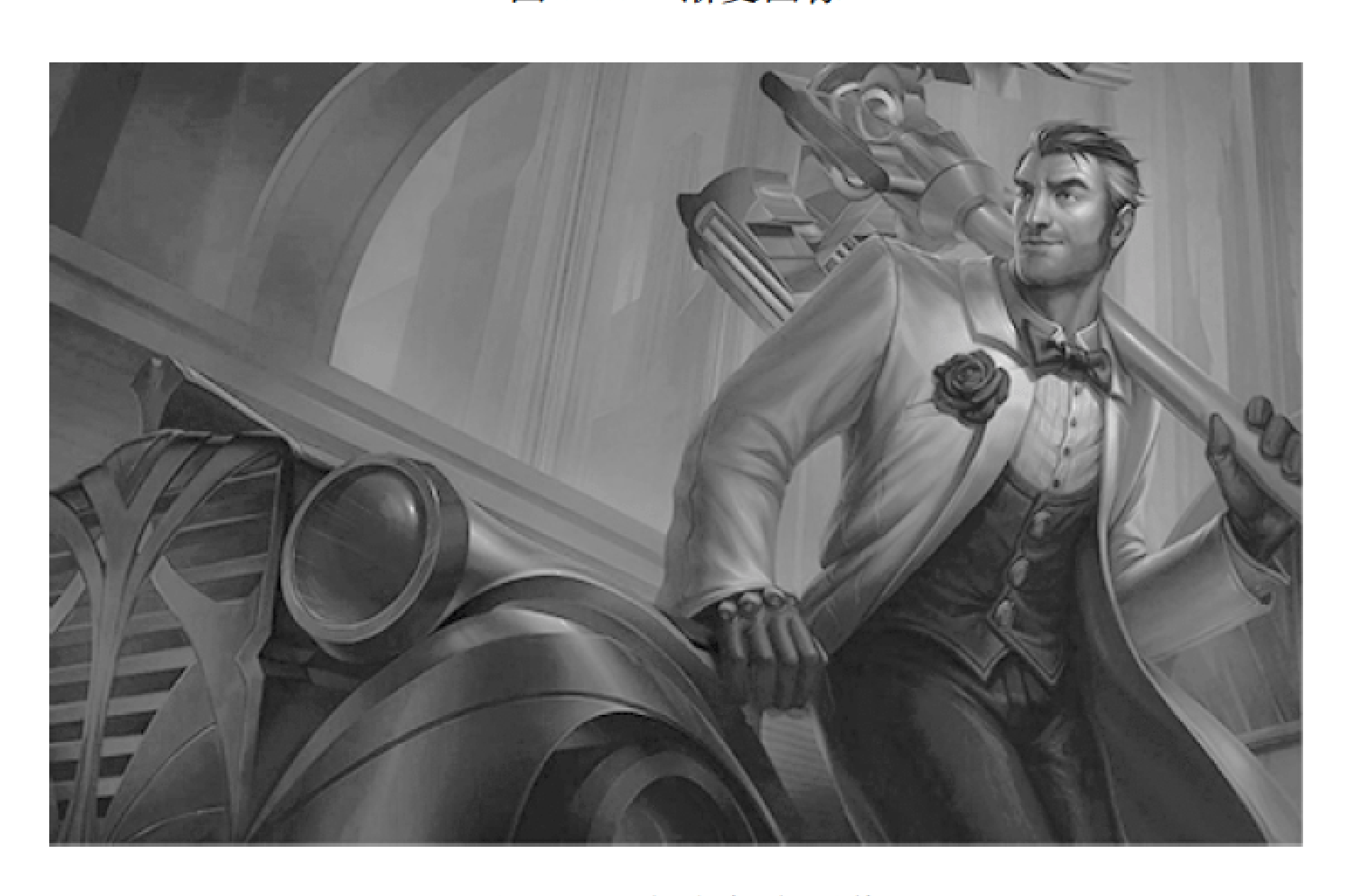


图 3-26　变成灰度图像

(3) 程序简析。

程序首先将一个三通道的彩色图像转变成三通道灰度图像，之后使用 for 循环，使两者不断地以不同的权重进行混合，并由新图像不断地覆盖旧图像，从而实现动态显示的过程，如图 3-27 所示。

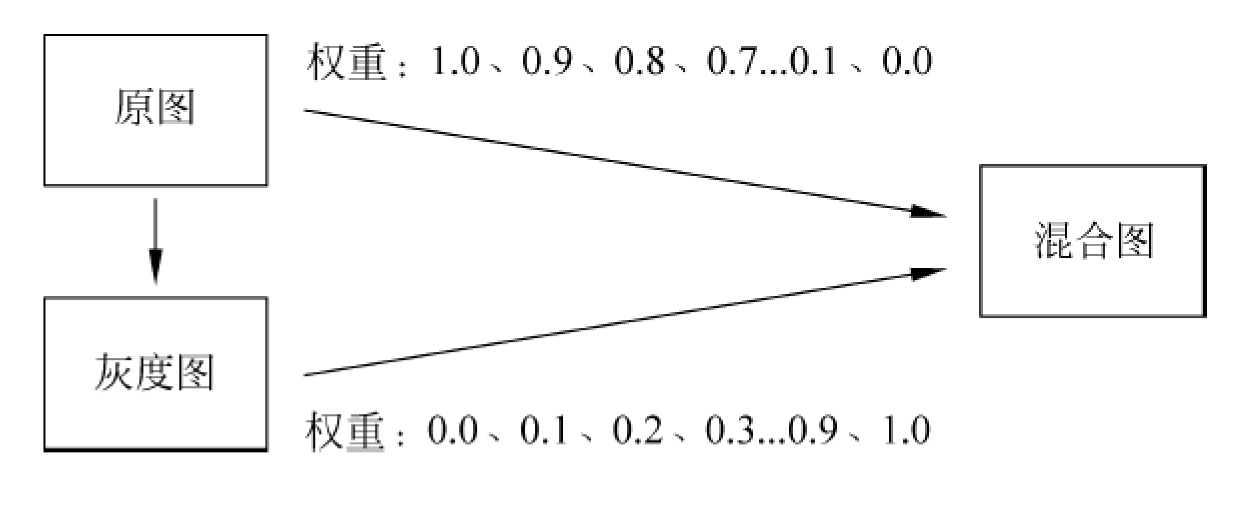


图 3-27　图像渐变原理图

头文件、计时函数、图像的读入输出部分不再重复，需要注意的是，使用 namedWindow() 函数可以实现图像的不断更新，如果直接使用 imshow() 函数则无法这样显示。

另外介绍一个图像变换函数 cvtColor(),该函数声明如下:

```
cvtColor(InputArray src, OutputArray dst, int code, int dstCn = 0);
```

- src

第一个参数 src 是 InputArray 类型的变量,是输入进来的要改变彩色空间的原始图像。

- dst

第二个参数 dst 是 OutputArrary 类型的变量,是将原图像做完彩色空间的转换之后,作为输出的图像。

- code

第三个参数 code 是 int 类型的变量,写入的是要转变的类型,例如程序中的第一个 code 为 CV_RGB2GRAY 代表将三通道 RGB 图像转换成单通道的灰度图像。程序中的第二个 code 为 CV_GRAY2RGB,是将单通道的灰度图像转换为三通道的 RGB 图像。

实现混合需要先得到一个三通道的灰度图像,才能和原图像进行线性混合,单通道的灰度图像不可以实现。但是空间转换中没有可以将三通道图像直接转换为三通道灰度图像的方法,所以采用的方案是先将三通道转换为单通道灰度,因为已经是灰度图,所以再次转换到三通道彩色图像时仍然不会带有任何颜色,通过这样的处理方式,就可以实现使用 addWeighted()函数进行线性混合。

空间转换命名的方式都比较简单,如刚刚说到的 CV_RGB2GRAY,就是将前半段的 RGB 图像转换成后半段中的 GRAY 图像,其他类型的命名也是同样道理。

这里将常用到颜色的空间转换总结如下:

```
RGB <--> BGR:
CV_BGR2BGRA、CV_RGB2BGRA、CV_BGRA2RGBA、CV_BGR2BGRA、CV_BGRA2BGR;
RGB <--> 5X5:
CV_BGR5652RGBA、CV_BGR2RGB555;
RGB <---> Gray:
CV_RGB2GRAY、CV_GRAY2RGB、CV_RGBA2GRAY、CV_GRAY2RGBA;
RGB <--> CIE XYZ:
CV_BGR2XYZ、CV_RGB2XYZ、CV_XYZ2BGR、CV_XYZ2RGB;
RGB <--> YCrCb(YUV) JPEG:
CV_RGB2YCrCb、CV_RGB2YCrCb、CV_YCrCb2BGR、CV_YCrCb2RGB、CV_RGB2YUV;
RGB <--> HSV:
CV_BGR2HSV、CV_RGB2HSV、CV_HSV2BGR、CV_HSV2RGB;
RGB <--> HLS:
CV_BGR2HLS、CV_RGB2HLS、CV_HLS2BGR、CV_HLS2RGB;
RGB <--> CIE L * a * b * :
CV_BGR2Lab、CV_RGB2Lab、CV_Lab2BGR、CV_Lab2RGB;
RGB <--> CIE L * u * v:
CV_BGR2Luv、CV_RGB2Luv、CV_Luv2BGR、CV_Luv2RGB;
RGB <--> Bayer:
CV_BayerBG2BGR、CV_BayerGB2BGR、CV_BayerRG2BGR、CV_BayerGR2BGR、CV_BayerBG2RGB、CV_BayerGB2RGB;
YUV420 <--> RGB:
CV_YUV420sp2BGR、CV_YUV420sp2RGB、CV_YUV420i2BGR、CV_YUV420i2RGB;
```

- dstCn

第四个参数 dstCn 是 int 型变量，如果不填则默认为 0，它是输出图像的通道数，若 dstCn=0，则表示与原图像的通道数保持一致。

3.4 图像去噪

图像信号在获取、传输和存储过程中，不可避免地会受到噪声的干扰，噪声降低了图像的质量，淹没了图像的边缘和细节特征，给图像分析和后续处理带来困难，图像噪声的消除是图像处理中的一个重要研究内容，能否有效地滤除噪声直接影响着图像后续工作的进行，因此图像去噪工作尤为重要[8]。本节将主要介绍 OpenCV 自带的均值滤波、高斯滤波、方框滤波这三种常用的去噪算法，并介绍一种去噪效果更好的非局部均值滤波。

3.4.1 均值滤波

1. 均值滤波算法的原理

均值滤波也称为线性滤波，其采用的主要方法为邻域平均法。线性滤波的基本原理是用均值代替原图像中的各个像素值，即为待处理的当前像素点(x,y)选择一个模板，该模板由其邻近的若干像素组成，求模板中所有像素的均值，再把该均值赋予当前像素点(x,y)，作为处理后图像在该点上的灰度值 $g(x,y)$，即

$$g(x,y)=\frac{\sum f(x,y)}{m} \tag{3-3}$$

式子中的 m 为该模板中包含当前像素在内的像素总个数。

均值滤波能够有效滤除图像中的加性噪声，但均值滤波本身存在着固有的缺陷，即它不能很好地保护图像细节，在图像去噪的同时也破坏了图像的细节部分，从而使图像变得模糊[9]。

2. 均值滤波的算法实现

```
#include<opencv2/opencv.hpp>
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
usingnamespace cv;
usingnamespace std;

int main()
{
    Mat image = imread("1.jpg");

    namedWindow("滤波前");
    namedWindow("滤波后");
    imshow("滤波前",image);

    Mat out;
    blur(image,out,Size(5,5));                              //均值滤波
```

```
        imshow("滤波后",out);

        waitKey();
        return 0;
    }
```

3. 程序运行结果

图 3-28 为原图像,图 3-29 和图 3-30 分别是均值滤波内核为 3×3 和 5×5 的滤波后的图像。



图 3-28 用于均值滤波的原始图像



图 3-29 内核为 3×3 的均值滤波结果



图 3-30 内核为 5×5 的均值滤波结果

4. 程序简析

上述程序调用了 OpenCV 内自带的均值滤波函数 blur(),该函数声明如下:

C++: void blur(InputArray src, OutputArray dst, Size ksize, Point anchor = Point(- 1, - 1), int borderType = BORDER_DEFAULT)

- src

第一个参数 src 是 InputArray 类型的变量,用于存放输入图像,使用 Mat 类的对象即可。该函数对通道独立处理,且可以处理任意通道数的图片。

- dst

第二个参数 dst 是 OutputArray 类型的变量，即目标图像，需要和原图像有一样的尺寸和类型。

- ksize

第三个参数 ksize 是 Size 类型的变量，即为内核的大小。通常的写法是 Size(w,h)表示内核的大小(w 为像素宽度，h 为像素高度)。Size(3,3)就表示 3×3 的核；同样，Size(5,5)就表示 5×5 的内核大小。

- anchor

第四个参数 anchor 是 Point 类型的变量，用来表示锚点(即被平滑的像素点)，注意其默认值为 Point(－1,－1)。如果点坐标是负值，就表示取核的中心为锚点，所以默认值 Point(－1,－1)表示这个锚点在方框核的中心。通常情况下，这个点都是核的中心点，而 anchor 这个参数也不需要填写。

- borderType

第五个参数 borderType 是 int 类型的变量，用于推断图像外部像素的某种边界模式。其默认值为 BORDER_DEFAULT，一般不需要去填写。

5. 程序优化

在图像处理尤其是图像去噪过程中，通常更改取值参数会比较麻烦，每一次都需要更改程序中的参数再运行查看效果。因此，可以通过创建一个轨迹条函数来解决这个问题，本节将简单说明如何使用轨迹条函数来方便查看均值滤波的效果。

程序如下：

```
#include<opencv2/opencv.hpp>
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include<iostream>
using namespace cv;
using namespace std;

Mat srcImage,dstImage;                                          //定义全局变量的输入和输出
int nBlur = 3;                                                  //起始的内核默认值

staticvoid MeanBlur(int,void * );

int main()
{

srcImage = imread("1.jpg");
namedWindow("原始图像");                                        //创建窗口
imshow("原始图像",srcImage);
namedWindow("均值滤波");
createTrackbar("内核值","均值滤波",&nBlur,20,MeanBlur);         //创建轨迹条
MeanBlur(nBlur,0);                                              //进行图像处理

waitKey();
```

```
return 0;
}

staticvoid MeanBlur(int,void * )
{
blur(srcImage,dstImage,Size(nBlur + 1,nBlur + 1));             //防止窗口为 0 崩溃
imshow("均值滤波",dstImage);
}
```

6. 实验结果

图 3-31 为输入的原图像；图 3-32 是内核大小为 1×1 的均值滤波后图像，即在不滤波情况下的输出图像；图 3-33 是内核大小为 3×3 的均值滤波后图像，图 3-34 是内核大小为 10×10 的均值滤波后图像。



图 3-31　均值滤波原始图像



图 3-32　内核为 1×1 的均值滤波结果



图 3-33　内核为 3×3 的均值滤波结果



图 3-34　内核为 10×10 的均值滤波结果

7. 程序简析

这段程序还是将图像进行均值滤波，唯一的区别在于增加了一个轨迹条函数：

createTrackbar()。

createTrackbar()声明如下：

```
int createTrackbar(conststring& trackbarname, conststring& winname, int * value, int count,
TrackbarCallbackonChange = 0,void * userdata = 0);
```

- trackbarname

第一个参数 trackbarname 是 conststring& 类型的变量，表示滑动线条部分的名称。

- winname

第二个参数 winname 是 conststring& 类型的变量，表示图像显示过程中的名称。如果前边有对应的 namedWindow 创建的图像窗口，那么这个名称就会在相应的图像上进行显示。

- value

第三个参数 value 是 int * 类型的变量，表示在滑动条中，起始时刻滑块的位置。

- count

第四个参数 count 是 int 类型的变量，表示滑动条中的最大值，即滑动条的上限。补充一点，在使用轨迹函数时，最小的滑动条位置是 0，这个是默认存在的，没有办法修改。

本程序中，功能函数为 blur(srcImage, dstImage, Size(nBlur＋1, nBlur＋1))，因为 count 的值最小可以取值到 0，但是在 Size(a, b)中，最小值必须大于 0，因此在这里使用了＋1 的方式，即滑块的位置＋1 即为内核的大小。

- onChange

第五个参数 onChange 是 TrackbarCallback 类型的变量，其自身的默认值为 0。这是一个指向回调函数的指针，每次滑块位置改变时，这个函数都会进行回调。并且这个函数的原型必须为 void XXXX(int, void *)。第一个参数是轨迹条的位置，第二个参数是用户数据。如果回调是 NULL 指针，表示没有使用回调函数，仅第三个参数 value 有变化。

- userdata

第六个参数 userdata 是 void * 型的变量，其自身的默认值为 0，表示用户传给回调函数的数据，用来处理轨迹函数。通常情况下不会更改这个参数，而是直接不填写这个参数，使用其默认值。

3.4.2 高斯滤波

1. 高斯滤波算法的原理

高斯滤波器(Gaussian Filter)是一种时频宽积最小的理想滤波器，有优良的特性。与传统的巴特沃思(Butterworth Filter)等滤波器有一整套成熟的设计理论和方法不同，高斯滤波器尚无完善的设计理论。不过高斯滤波克服了传统滤波相移和设计复杂的缺陷，因此在图像处理、计算机视觉、通信技术、计量测试、时频分析、小波变换等众多领域得到了广泛的应用[10]。

高斯滤波器的脉冲响应函数为：

$$h(x) = \frac{1}{\alpha\lambda_c}\exp\left[-\pi\left(\frac{x}{\alpha\lambda_c}\right)^2\right] \tag{3-4}$$

式子中 $\alpha=\sqrt{\frac{\ln 2}{\pi}}$；$\lambda_c$ 为滤波器的截止波长。

通过一次卷积运算可以将原始信号 $z(x)$ 分离成为低频信号 $W(x)$ 和高频信号 $R(x)$ 两个部分：

$$W(x)=\int_{-\infty}^{+\infty} h(x-\varepsilon)\cdot z(x)\mathrm{d}\varepsilon \tag{3-5}$$

$$R(x)=z(x)-W(x) \tag{3-6}$$

2. 算法实现

```
#include<opencv2/opencv.hpp>
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include<iostream>
usingnamespace cv;
usingnamespace std;

Mat srcImage,dstImage;
int nGaussian = 3;

staticvoid Gaussian(int,void * );

int main()
{
    srcImage = imread("1.jpg");
    namedWindow("原始图像");
    imshow("原始图像",srcImage);

    namedWindow("高斯滤波");
    createTrackbar("内核值:","高斯滤波",&nGaussian,20,Gaussian);
    Gaussian(nGaussian,0);

    waitKey();
    return 0;
}

staticvoid Gaussian(int,void * )
{
    GaussianBlur(srcImage,dstImage,Size(nGaussian * 2 + 1,nGaussian * 2 + 1),0,0);
    imshow("高斯滤波",dstImage);
}
```

3. 实验结果

图 3-35 为原图像，图 3-36 为滤波后参数值取 3 的图像，图 3-37 为图像滤波后参数值为 7 的图像。

4. 程序简析

本程序与前面的均值滤波程序很相似，只是换了一个图像处理函数。除了高斯滤波，OpenCV 中还有很多去噪函数，例如方框滤波、中值滤波等，都是这种应用方式，仅仅是换了一个函数而已，此处不再赘述。



图 3-35　用于高斯滤波的原始图像



图 3-36　高斯滤波内核为 3 的图像



图 3-37　高斯滤波内核为 7 的图像

本次使用的是 GaussianBlur()函数，其声明如下：

```
void GaussianBlur(InputArray src, OutputArray dst, Size ksize, double sigmaX, double sigmaY = 0,
int borderType = BORDER_DEFAULT );
```

- src

第一个参数 src 是 InputArray 类型的变量，其用处是存储 Mat 类的需要进行高斯滤波的原始图像图像数据。

- dst

第二个参数 dst 是 OutputArray 类型的变量，其用处是存储 Mat 类的高斯滤波之后的图像数据。

- ksize

第三个参数 ksize 是 Size 型的变量，表示高斯滤波器的模板大小。

- sigmaX、sigmaY

第四个参数 sigmaX 和第五个参数 sigmaY 都是 double 类型的变量，两者分别表示高斯滤波在横向和纵向的滤波系数。

- borderType

第六个参数是 int 型的变量，表示边缘检测点的插值类型。

3.4.3　非局部均值滤波

1. 非局部均值算法的原理

非局部均值算法是一种图像去噪算法，而领域平均去噪是非局部均值算法的理论基础，

接下来先介绍邻域平均去噪算法的原理，再介绍非局部均值算法的理论。

1）邻域平均去噪算法[11]

假设图像受到加性高斯白噪声的干扰，那么可以得到被干扰后的图像：

$$g(x) = f(x) + \varphi(x) \tag{3-7}$$

其中 $f(x)$表示原本的纯净图像，$g(x)$表示受到干扰之后的输出的图像，$\varphi(x)$表示均值为零的高斯白噪声。

邻域平均的思想是较早提出的一种去噪理念，其利用邻域像素的均值估计中心像素点。该方法是基于这样一种前提假设：噪声在图像局部区域内服从相同的分布，并且像素点的灰度值在非常小的范围内是缓慢变化的，即一定程度上是相似的。因此，可以用邻域像素估计中心像素的值。对于一个给定的像素点 i，设 $N(i)$为所选取的用于平均计算的邻域，则像素 $f(i)$的估计值$\hat{f}(i)$为：

$$\hat{f}(i) = \frac{1}{N}\sum_{j\in N(i)}(f(j)+\varphi(j))N(i) = \frac{1}{N}\sum_{j\in N(i)}f(j) + \frac{1}{N}\sum_{j\in N(i)}\varphi(j) \tag{3-8}$$

式中 N 表示 $N(i)$内像素点的个数，因 $E[\varphi(j)]=0$，若 $f(i)=f(j)$，则有$\hat{f}(i)=\hat{f}(j)$。用 VA 表示像素点 i 去噪后的方差，σ^2 为噪声信号 $\varphi(j)$的方差，则有：

$$\mathrm{VA} = \mathrm{Var}\left\{\frac{1}{N}\sum_{j\in N(i)}\varphi(j)\right\} = \frac{1}{N^2}\sum_{j\in N(i)}\mathrm{Var}[\varphi(j)] = \frac{1}{N^2}N\sigma^2 = \frac{1}{N}\sigma^2 \tag{3-9}$$

由式(3-9)可知，N 值越大，滤波后的像素 i 处的噪声方差就会越小，仅为原来的$\frac{1}{N}$。

在实际去噪过程中，由于噪声的污染，在噪声图像中寻找 $f(i)$与真实值完全相同的像素比较困难，并且在噪声较大时去噪能力有限。虽然完全相同的像素较少，但是在一定邻域内，相似的像素却有很多。为了能够充分利用这一特性，学者们提出了加权的思想。

加权平均的思想利用图像中的自相似信息，根据像素之间的相似程度设置权值的大小。基于加权平均思想的邻域平均去噪算法取得了非常好的去噪效果。此类算法的关键在于如何度量像素之间的相似性或者构造权值函数，这就是非局部均值算法的前身。

2）非局部均值算法

局部去噪和变换域去噪算法在去除噪声的同时，能够恢复图像的主要几何结构信息，但在精细结构、细节信息和纹理的保留上明显不足。图像中的任何一个像素都不是孤立的，而是与其周围的像素点结合在一起共同构成图形的几何结构。以某一个像素点为中心的窗口邻域，可以很好地描述像素点的结构特征。针对任何一个像素点图像块的所有集合可以看作是图像的一种过完备表示。它采用的结构相似性定义像素间的差异，并对像素周围整个区域的灰度分布做整体对比，根据图像中灰度分布的相似性决定权值的大小，如图 3-38 所示，假设 p、q_1、q_2、q_3 具有完全相同的灰度值，那么 q_1、q_2、q_3 三者都会根据与 p 点不同的欧氏距离而得到相应的权值，但 q_1 和 q_2 的邻域灰度分布与 p 更接近，因此贡献更大的权值，q_3 则对 p 贡献较小的权值[12]。

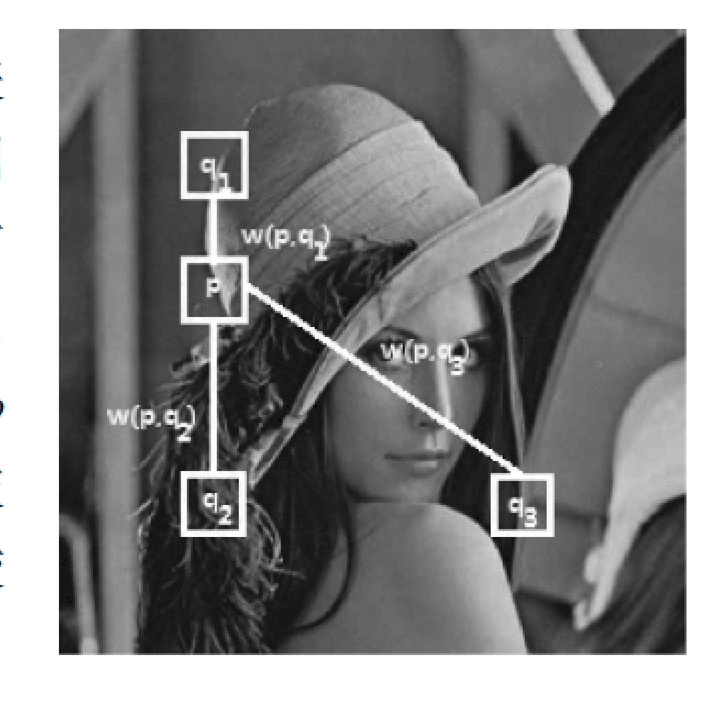


图 3-38 NLM 算法原理



假设一幅含噪图像 $z=\{z(i)\mid i\in I\}$，其定义在有界域

$I \in N^2$。在这幅图像中,对于某个像素点 i,非局部均值滤波算法利用所有的像素的加权平均来得到该点的估计值 $NL(z)(i)$,即:

$$NL(z)(i) = \sum_{j \in I} w(i,j) z(j) \tag{3-10}$$

其中,权值$\{w(i,j)\}_j$ 依赖于像素 i 与像素 j 之间的相似性,且满足如下条件:$0 \leqslant w(i,j) \leqslant 1$ 且 $\sum_j w(i,j) = 1$。

图像域 I 上的邻域系统 $N = \{N_i\}_{i \in I}$是图像域 I 的子集,使得对于所有的像素点 $i \in I$ 都必须满足以下两个条件:

- $i \in N_i$;
- $j \in N_i \Rightarrow i \in N_j$;

其中,N_i 是像素 i 的窗口邻域。

为了能够更好地适应图像不同区域的特征,可以将相似性窗口取不同的形状和大小。为了方便起见,这里使用固定大小的方形窗口。相似性窗口 N_i 内的灰度值向量可以表示如下:

$$z(N_i) = (z(j), j \in N_i) \tag{3-11}$$

灰度值向量 $z(N_i)$和 $z(N_j)$之间的相似性可以用来决定像素点 i 和像素点 j 之间的相似性,即在加权平均时,那些与 $z(N_i)$具有相似灰度值向量的像素点将被分配较大的权值,反之则被分配到较小的权值。为了能够定量地计算 $z(N_i)$和 $z(N_j)$之间的相似性,可以采用高斯加权的欧氏距离 $\| z(N_i) - z(N_j) \|_{2,a}^2$。在含噪声的图像与滤波后的图像对应位置的窗口内,灰度值向量之间的欧氏距离满足如下关系:

$$E \| z(N_i) - z(N_j) \|_{2,a}^2 = \| y(N_i) - y(N_j) \|_{2,a}^2 + 2\sigma^2 \tag{3-12}$$

其中,z 与 y 分别表示带有噪声图像与滤波后的图像,σ^2 是噪声的方差。

基于以上的式子,可以得到像素点 i 和像素点 j 之间的权值 $w(i,j)$:

$$w(i,j) = \frac{1}{Z(i)} \exp\left(\frac{- \| z(N_i) - z(N_j) \|_{2,a}^2}{h^2} \right) \tag{3-13}$$

其中 $Z(i) = \sum_j \exp\left(\frac{- \| z(N_i) - z(N_j) \|_{2,a}^2}{h^2} \right)$是归一化常数;$\| z(N_i) - z(N_j) \|_{2,a}^2$是指 i 块和 j 块的加权欧式距离的平方,用 $d(i,j)$来表示,$a(a>0)$是指高斯核的标准差,由选定像素邻域的窗口大小决定。参数 h 控制指数函数的衰减速度,同时影响着权值的衰减速度,$h = c \times \sigma$。c 是用于调整的系数,Buades 将其范围规定在 1~10 之间。

最终可以将非局部均值滤波的算法整理为如下 3 个算式:

$$d(i,j) = \| z(N_i) - z(N_j) \|_{2,a}^2 \tag{3-14}$$

$$w(i,j) = \exp\left(- \frac{d(i,j)}{h^2} \right) \tag{3-15}$$

$$z(i) = \frac{\sum_{j \in I} w(i,j) z(j)}{\sum_{j \in I} w(i,j)} \tag{3-16}$$

图 3-39 显示了非局部均值滤波算法的执行的过程。在算法执行的过程中,需要设置两个窗口的大小:一个是像素邻域窗口尺寸 $K \times K$,一个是像素邻域窗口搜索范围的窗口尺寸 $L \times L$,

即在 $L\times L$ 大小的窗口内选择像素的邻域大小为 $K\times K$ 执行非局部均值滤波算法，$K\times K$ 的窗口在 $L\times L$ 的区域内滑动，根据区域的相似性确定区域中心像素灰度的贡献权值，而在这里的图像处理实现中，窗口尺寸为 $K=7$，滑动范围为 $L=17$。

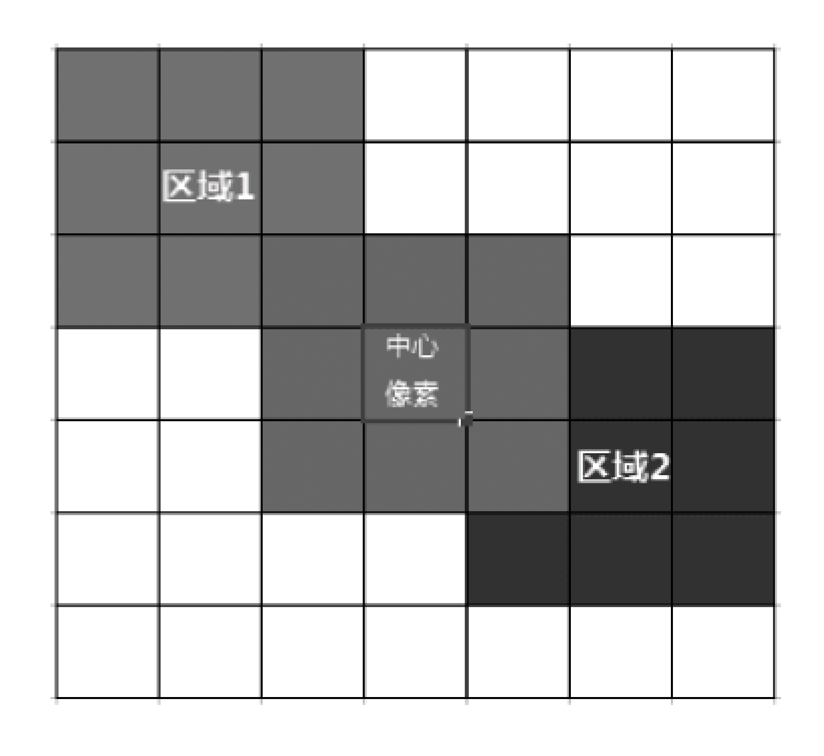


图 3-39 NLM 算法执行示意图

在相对稳定的条件下（即图像有足够大的尺寸时），对于图像内部的各种细节都能找到足够多的相似区域。

2. NLM 算法实现

```
#include<opencv2/opencv.hpp>
#include<iostream>
usingnamespace cv;
usingnamespace std;

void addNoiseSoltPepperMono(Mat& src, Mat& dest,double per)
{
    cv::RNG rng;
#pragma omp parallel for
for(int j=0;j<src.rows;j++)
    {
        uchar* s=src.ptr(j);
        uchar* d=dest.ptr(j);
for(int i=0;i<src.cols;i++)
        {
double a1 = rng.uniform((double)0, (double)1);

if(a1>per)
                d[i]=s[i];
else
            {
double a2 = rng.uniform((double)0, (double)1);
if(a2>0.5)d[i]=0;
else d[i]=255;
            }
        }
    }
}
void addNoiseMono(Mat& src, Mat& dest,double sigma)
{
    Mat s;
    src.convertTo(s,CV_16S);
    Mat n(s.size(),CV_16S);
    randn(n,0,sigma);
    Mat temp = s+n;
    temp.convertTo(dest,CV_8U);
}
```

```
void addNoise(Mat&src, Mat& dest, double sigma,double sprate = 0.0)
{
if(src.channels() == 1)
    {
        addNoiseMono(src,dest,sigma);
if(sprate!= 0)addNoiseSoltPepperMono(dest,dest,sprate);
return;
    }
else
    {
        vector<Mat> s;
        vector<Mat> d(src.channels());
        split(src,s);
for(int i = 0;i<src.channels();i++)
        {
            addNoiseMono(s[i],d[i],sigma);
if(sprate!= 0)addNoiseSoltPepperMono(d[i],d[i],sprate);
        }
        cv::merge(d,dest);
    }
}

staticdouble getPSNR(Mat& src, Mat& dest)
{
int i,j;
double sse,mse,psnr;
    sse = 0.0;
for(j = 0;j<src.rows;j++)
    {
        uchar * d = dest.ptr(j);
        uchar * s = src.ptr(j);
for(i = 0;i<src.cols;i++)
        {
            sse += ((d[i] - s[i]) * (d[i] - s[i]));
        }
    }
if(sse == 0.0)
    {
return 0;
    }
else
    {
        mse = sse /(double)(src.cols * src.rows);
        psnr = 10.0 * log10((255 * 255)/mse);
return psnr;
    }
}

double calcPSNR(Mat& src, Mat& dest)
{
    Mat ssrc;
```

```
    Mat ddest;
if(src.channels() == 1)
    {
        src.copyTo(ssrc);
        dest.copyTo(ddest);
    }
else
    {
        cvtColor(src,ssrc,CV_BGR2YUV);
        cvtColor(dest,ddest,CV_BGR2YUV);
    }
double sn = getPSNR(ssrc,ddest);
return sn;
}

void nonlocalMeansFilter(Mat& src, Mat& dest, int templeteWindowSize, int searchWindowSize,
double h, double sigma = 0.0)
{
if(dest.empty())dest = Mat::zeros(src.size(),src.type());
    constint tr = templeteWindowSize >> 1;
constint sr = searchWindowSize >> 1;
constint bb = sr + tr;
    constint D = searchWindowSize * searchWindowSize;
    constint H = D/2 + 1;
constdouble div = 1.0/(double)D;
constint tD = templeteWindowSize * templeteWindowSize;
constdouble tdiv = 1.0/(double)(tD);
    Mat im;
    copyMakeBorder(src,im,bb,bb,bb,bb,cv::BORDER_DEFAULT);
    vector < double > weight(256 * 256 * src.channels());
    double * w = &weight[0];
constdouble gauss_sd = (sigma == 0.0) ? h :sigma;
double gauss_color_coeff = - (1.0/(double)(src.channels())) * (1.0/(h * h));
for(int i = 0; i < 256 * 256 * src.channels(); i++)
    {
double v = std::exp( max(i - 2.0 * gauss_sd * gauss_sd,0.0) * gauss_color_coeff);
        w[i] = v;
    }

constint cstep = im.step - templeteWindowSize * 3;
constint csstep = im.step - searchWindowSize * 3;
for(int j = 0;j < src.rows;j++)
        {
            uchar * d = dest.ptr(j);
int * ww = newint[D];
double * nw = newdouble[D];
for(int i = 0;i < src.cols;i++)
            {
double tweight = 0.0;
                uchar * tprt = im.data + im.step * (sr + j) + 3 * (sr + i);
                uchar * sptr2 = im.data + im.step * j + 3 * i;
```

```
                    for(int l = searchWindowSize,count = D - 1;l -- ;)
                        {
                            uchar * sptr = sptr2 + im.step * (l);
        for (int k = searchWindowSize;k -- ;)
                            {

        int e = 0;
                                uchar * t = tprt;
                                uchar * s = sptr + 3 * k;
        for(int n = templeteWindowSize;n -- ;)
                                {
        for(int m = templeteWindowSize;m -- ;)
                                    {
              e += (s[0] - t[0]) * (s[0] - t[0]) + (s[1] - t[1]) * (s[1] - t[1]) + (s[2] - t[2]) * (s[2] - t[2]);
                                        s += 3,t += 3;
                                    }
                                    t += cstep;
                                    s += cstep;
                                }
        constint ediv = e * tdiv;
                                ww[count -- ] = ediv;
                                tweight += w[ediv];
                            }
                        }
        if(tweight == 0.0)
                        {
        for(int z = 0;z < D;z++) nw[z] = 0;
                            nw[H] = 1;
                        }
        else
                        {
        double itweight = 1.0/(double)tweight;
        for(int z = 0;z < D;z++) nw[z] = w[ww[z]] * itweight;
        double r = 0.0,g = 0.0,b = 0.0;
                        uchar * s = im.ptr(j + tr); s += 3 * (tr + i);
        for(int l = searchWindowSize,count = 0;l -- ;)
                        {
        for(int k = searchWindowSize;k -- ;)
                            {
                                r += s[0] * nw[count];
                                g += s[1] * nw[count];
                                b += s[2] * nw[count++];
                                s += 3;
                            }
                            s += csstep;
                        }
                        d[0] = saturate_cast<uchar>(r);
                        d[1] = saturate_cast<uchar>(g);
                        d[2] = saturate_cast<uchar>(b);
                        d += 3;
                        }
```

```
                }
    delete[] ww;
    delete[] nw;
            }

}

int main(int argc, char * * argv)
{
constdouble noise_sigma = 15.0;

    Mat src = imread("lena512.jpg",1);

    Mat snoise;
    Mat dest;
    addNoise(src,snoise,noise_sigma);
    int64 pre = getTickCount();
    pre = getTickCount();
    nonlocalMeansFilter(snoise,dest,3,7,noise_sigma,noise_sigma);

    cout <<"time: "<< 1000.0 * (getTickCount() - pre)/(getTickFrequency())<<" ms"<< endl;
    cout <<"nonlocal: "<< calcPSNR(src,dest)<< endl << endl;
    imwrite("nonlocal.png",dest);
    imshow("noise", snoise);
imshow("Non - local Means Filter", dest);
    waitKey();
return 0;
}
```

3. 程序运行结果

如图 3-40 所示，左图为原图像，中间图为加噪后图像，右边为 NLM 算法去噪后的图像。



图 3-40　NLM 算法结果

4. NLM 算法的 MATLAB 实现

MATLAB 程序相比于 C++更好理解，接下来将给出一套 NLM 算法使用 MATLAB 语言实现的程序，但该程序只能处理单通道的灰度图像，程序如下：

```
clear all
```

```
close all
n = 3;
sigma = 8;
M = 8;
T = sigma ^2;
y00 = imread('lena64.jpg');
y0 = y00(1:64,1:64);
y0 = double(y0);
N = size(y0,1);
noise = randn(N,N);
y = y0 + sigma * noise;
tic//计时代码 tic toc
figure(1);
clf;
image([y0,y]);
colormap(gray(256));
axis image;
axis off;
drawnow;
yout = zeros(N,N);
h = waitbar(0,'NLM filtering ...');
y = [y(:,M + n: - 1:1),y,y(:,end - M - n + 1:end)];
y = [y(M + n: - 1:1,:); y; y(end - M - n + 1:end,:)];
for i = n + M + 1:1:N + M + n
    waitbar((i - M - n)/N);
for j = n + M + 1:1:N + M + n
        Center = y(i - n:i + n,j - n:j + n);
Weights = zeros(2 * M + 1,2 * M + 1);
for p = - M:M
for q = - M:M
                Patch = y(i + p - n:i + p + n,j + q - n:j + q + n);
                dist2 = mean((Patch(:) - Center(:)).^2);
                Weights(p + M + 1,q + M + 1) = exp( - dist2/T);
end;
end;
        Weights = Weights/sum(Weights(:));
        yout(i - n - M,j - n - M) = sum(sum(y(i - M:i + M,j - M:j + M). * Weights));
end;
end;
close(h)
y = y(M + n + 1:M + n + N,M + n + 1:M + n + N);
figure(2);
clf;
image([y0,y,yout]);
colormap(gray(256));
axis image;
axis off;
drawnow;
toc
```

3.5　双目视觉测量物体深度

在机器视觉领域中，双目视觉是一种应用很广泛的手段。双目视觉利用两台摄像机同时对物体进行拍摄，根据景物点在左右摄像机图像上位置关系，可以计算出景物点的三维坐标，从而可以实现三维测量和恢复。双目视觉测量系统因其结构简单、操作方便、成本低，以及具有在线、实时测量的潜力，而被广泛应用于机器人指导、工业生产现场以及航空等诸多领域[13]。



图 3-41　双目摄像头

如图 3-41 所示为常见的双目摄像头。

3.5.1　双目视觉原理

双目视觉测量的系统模型如图 3-42 所示，设左侧摄像机坐标为 $o_1x_1y_1z_1$，右侧摄像机坐标为 $o_2x_2y_2z_2$，选取左侧摄像机坐标系为世界坐标系，左侧理想图像坐标系为 $O_1X_1Y_1$，右侧理想图像坐标为 $O_2X_2Y_2$，f_1 和 f_2 分别为左右摄像机的焦距，像元尺寸为 w_1 和 w_2，则由空间几何关系可以得到空间点 P 在测量坐标系下的三维坐标为：

$$\begin{cases} X_w = \dfrac{B\cot(\omega_1+\alpha_1)}{\cot(\omega_1+\alpha_1)+\cot(\omega_2+\alpha_2)} \\ Y_w = Y_1\dfrac{z\sin\omega_1}{f_1\sin(\omega_1+\alpha_1)} = Y_2\dfrac{z\sin\omega_2}{f_2\sin(\omega_2+\alpha_2)} \\ Z_w = \dfrac{B}{\cot(\omega_1+\alpha_1)+\cot(\omega_2+\alpha_2)} \end{cases} \tag{3-17}$$

式子中：$\omega_1=\arctan(X_1/f_1)$，$\omega_2=\arctan(X_2/f_2)$，$z$ 为物距，B 为系统基线距[14]。

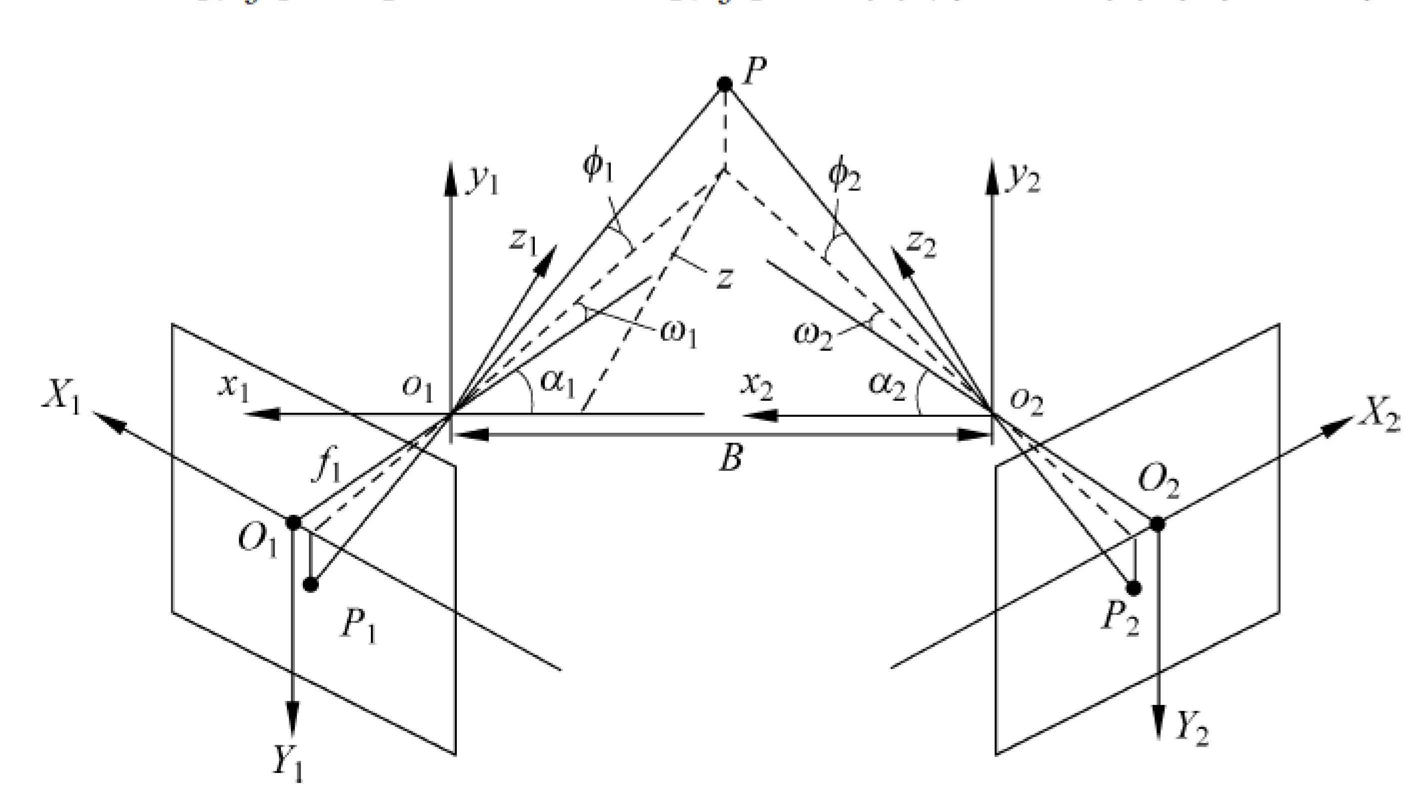


图 3-42　双目视觉原理图

3.5.2　双目视觉标定

双目视觉可以确定物体的空间坐标，还可以通过这种方式来测量物体距离，也就是物体和摄像头之间的距离，这也被称作测量物体的深度，实际测量中需要注意很多现实的问题。



1. 预处理

在理论上,双目摄像机的两个摄像头即便焦距不同也可以实现双目视觉测量物体的深度,但是实际上在操作时务必使用两个焦距相同的摄像头。如果是使用两个单目摄像头拼装成的双目摄像头,请尽量固定好两个摄像头之间的距离,并且保证两个单目摄像头不会有高度差,尽可能地保持水平等。通常如果使用了集成好的双目摄像头,这样拍出的两个图像会集成在一个图片上,这种图像则需要进行一次图像分割预处理。如果是两个单独的摄像头,或者是双目摄像头拍出的两个图像是分开进行存储的,一定要注意每次拍摄好的图像要分别重命名,如 left1 和 right1,否则很容易在处理之前发生两个图像不配套的情况。

2. 摄像机标定

在预处理结束之后,就需要对摄像机进行标定,为后续的双目摄像头测深度做好预处理。标定相当于使用一些手段获取到双目摄像头的焦距、双目摄像头之间的距离等信息。因此不论是使用 MATLAB 工具箱进行标定,还是使用 OpenCV 标定甚至是手动输入都是可以的,不过 MATLAB 工具箱标定的精度更高。本节将介绍两种使用 MATLAB 工具箱进行标定的方法,OpenCV 标定则在后面的程序实现中进行介绍。

方法一:calib 工具箱

calib 工具箱不是 MATLAB 自身集成进去的工具箱,但是其标定的效果通常比其他标定方法好,因此这种标定方式还在普遍使用中,下面将介绍如何使用 calib 工具箱进行标定[16]。

(1) MATLAB 和 calib 的下载。

MATLAB 的版本没有限制,后续的标定将以 MATLAB2016b 来作示例,calib 工具箱全称是 TOOLBOX_calib。

下载地址:http://www.vision.caltech.edu/bouguetj/calib_doc/

CSDN 下载地址:http://download.csdn.net/download/kevinfrankchen/9454023

百度网盘:链接:http://pan.baidu.com/s/1c2Eurc 密码:l9x8

(2) 解压并安装。

下载完成后需要将其解压,将文件夹名称修改为 calib,并放在 MATLAB 的默认路径下或者是放在 MATLAB 的 toolbox 文件夹中。示例中的路径为:

```
E:\MATLAB2015B\toolbox
```

解压后的效果如图 3-43 所示。

(3) 在 MATLAB 中添加 calib 工具箱。

将 calib 放进指定的路径之后,就将 MATLAB 打开,在“主页”界面中,找到“设置路径”选项,并打开,如图 3-44 所示。

进入添加路径界面之后单击“添加并包含子文件夹…”按钮,添加路径和对应路径下的子文件夹,如图 3-45 所示。

找到之前放置的 calib 文件夹,单击“选择文件夹”将其添加至路径中,如图 3-47 所示。最后回到如图 3-46 所示的界面中,单击“保存”按钮即可。至此,calib 标定工具箱添加完成。



图 3-43　calib 安放位置



图 3-44　MATLAB 添加工具箱第一步

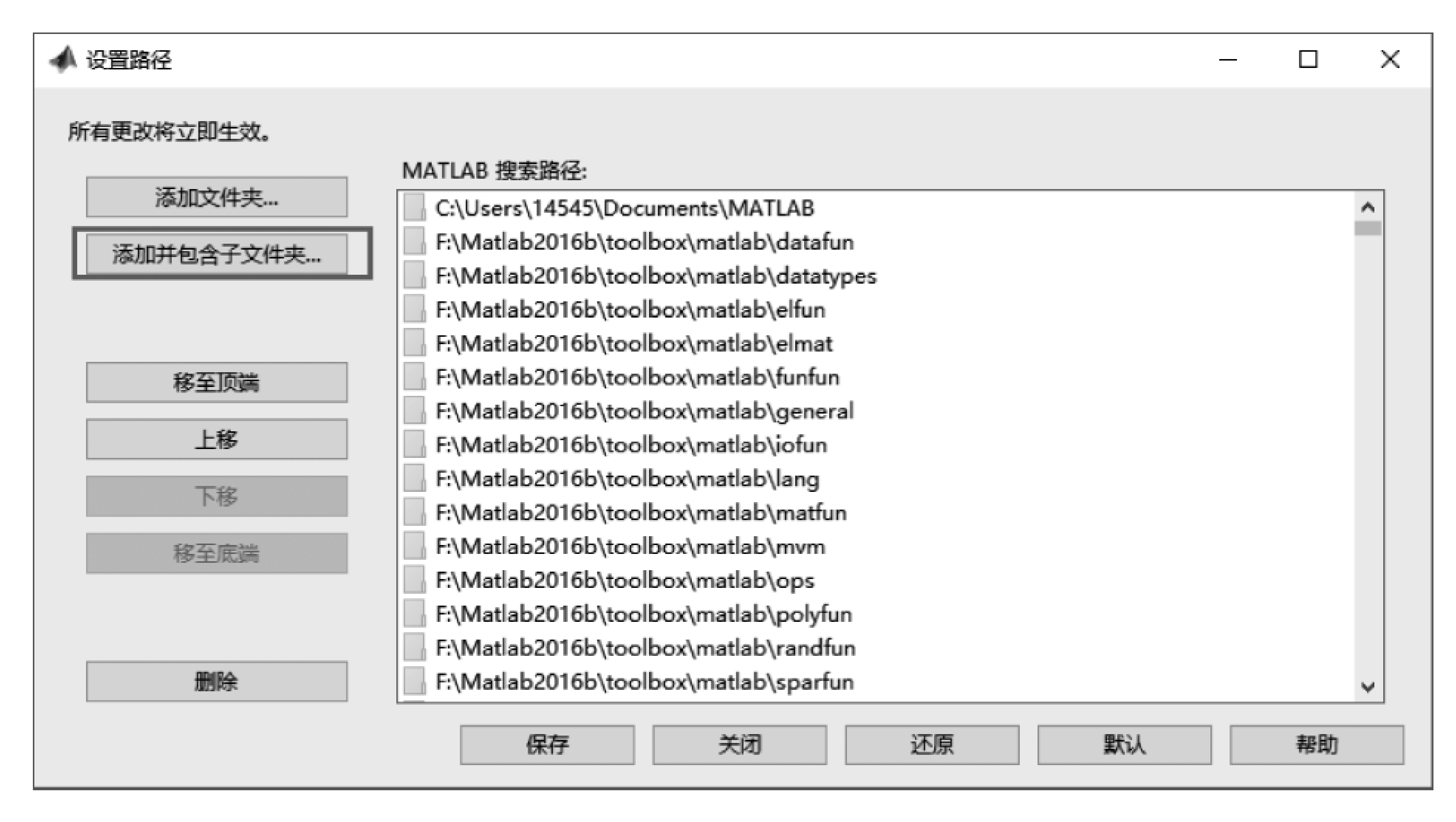


图 3-45　MATLAB 添加工具箱第二步

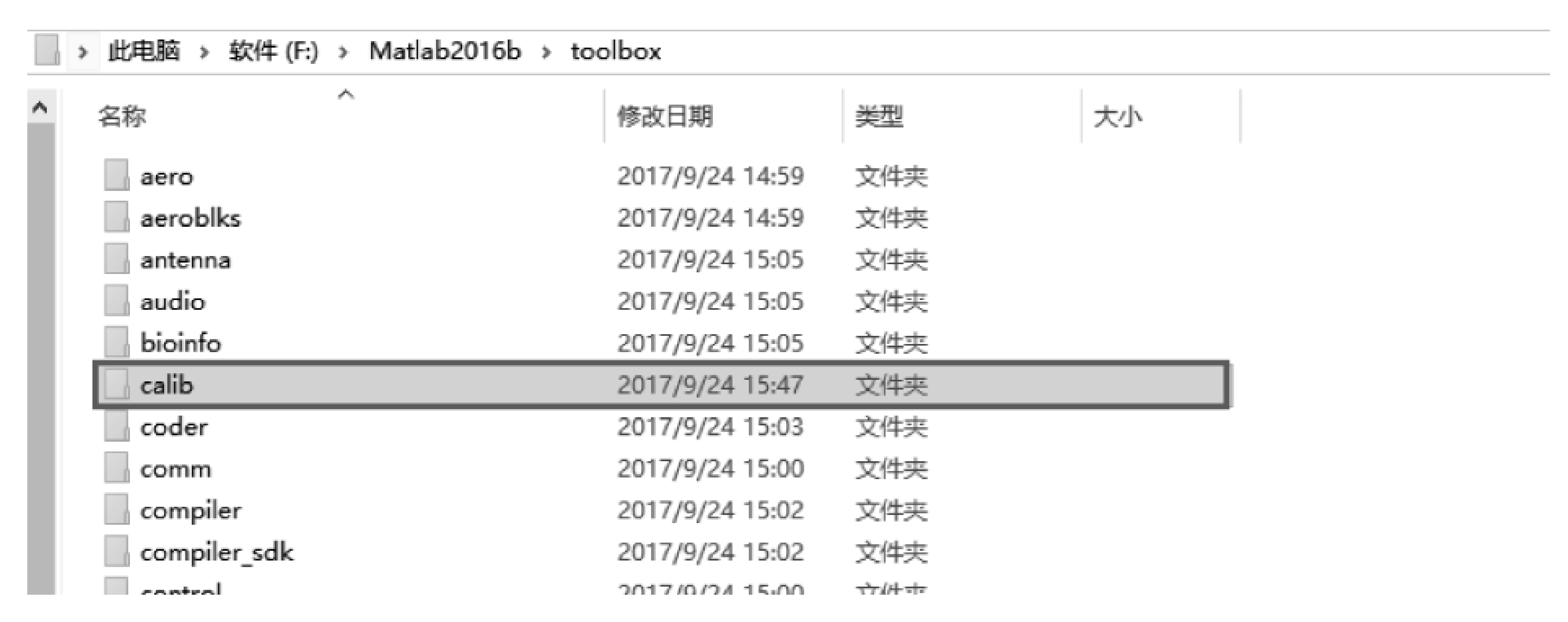


图 3-46　MATLAB 添加工具箱第三步

（4）在 MATLAB 中添加图像路径。

首先将左边相机拍到的图像按照拍照的先后顺序排列好，并依次命名为“left_＋数字”的形式，如 left_1. bmp、left_2. bmp。将右摄像头拍到的图像依次命名为“right_＋数字”的形式。最后将两个摄像头拍到的图像分别放在两个文件夹中，路径文件尽量不包含中文，虽然高版本的 MATLAB 可以兼容中文路径，但是仍然可能出现错误。如图 3-47 和图 3-48 所示为双目摄像头分别拍摄到的图像。为了演示方便，将左摄像头取出前十张图像命名为 left_1. bmp、left_2. bmp、…、left_10. bmp。同样，右摄像头取出与左摄像头对应的十张图像也分别命名为 right_1. bmp、right_2. bmp、…、right_10. bmp。

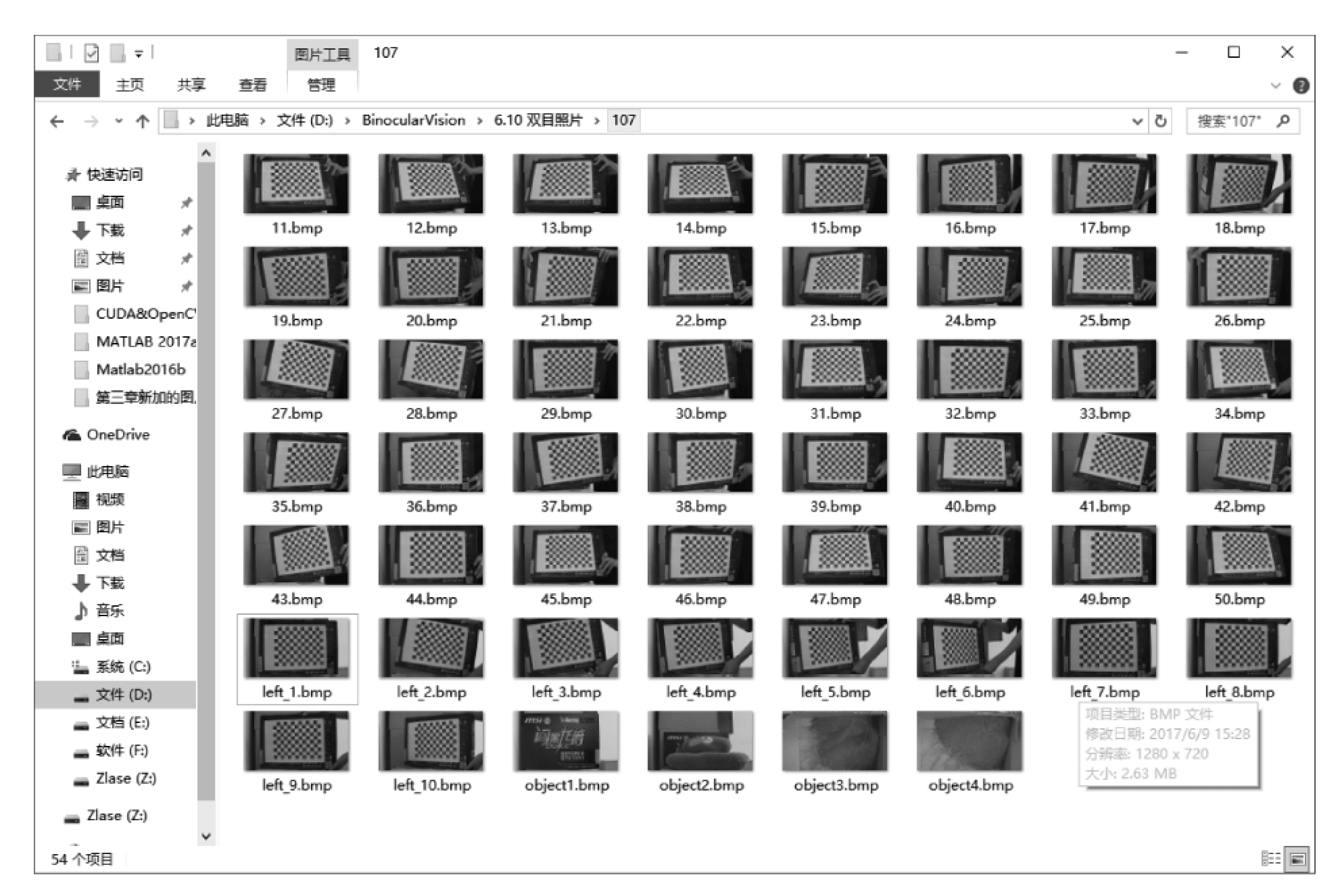


图 3-47　左摄像头拍到的图像

两个文件夹的路径分别为：

```
D:\BinocularVision\6.10_双目照片\107
D:\BinocularVision\6.10_双目照片\108
```

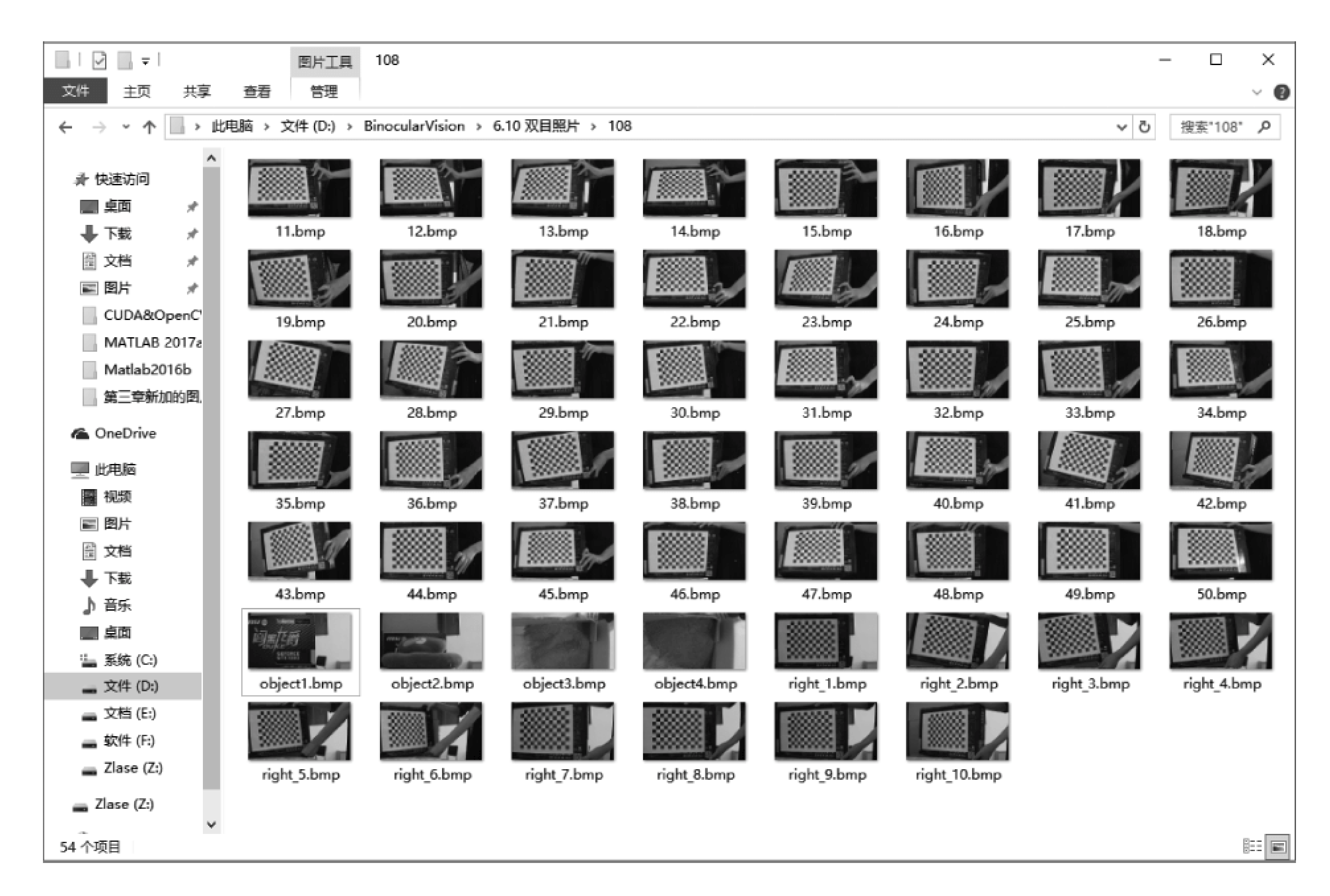


图 3-48　右摄像头拍到的图像

之后打开 MATLAB 中的“设置路径”界面，并单击“添加并包含子文件夹…”按钮，将两个文件夹添加到 MATLAB 的 path 中，如图 3-49 所示。

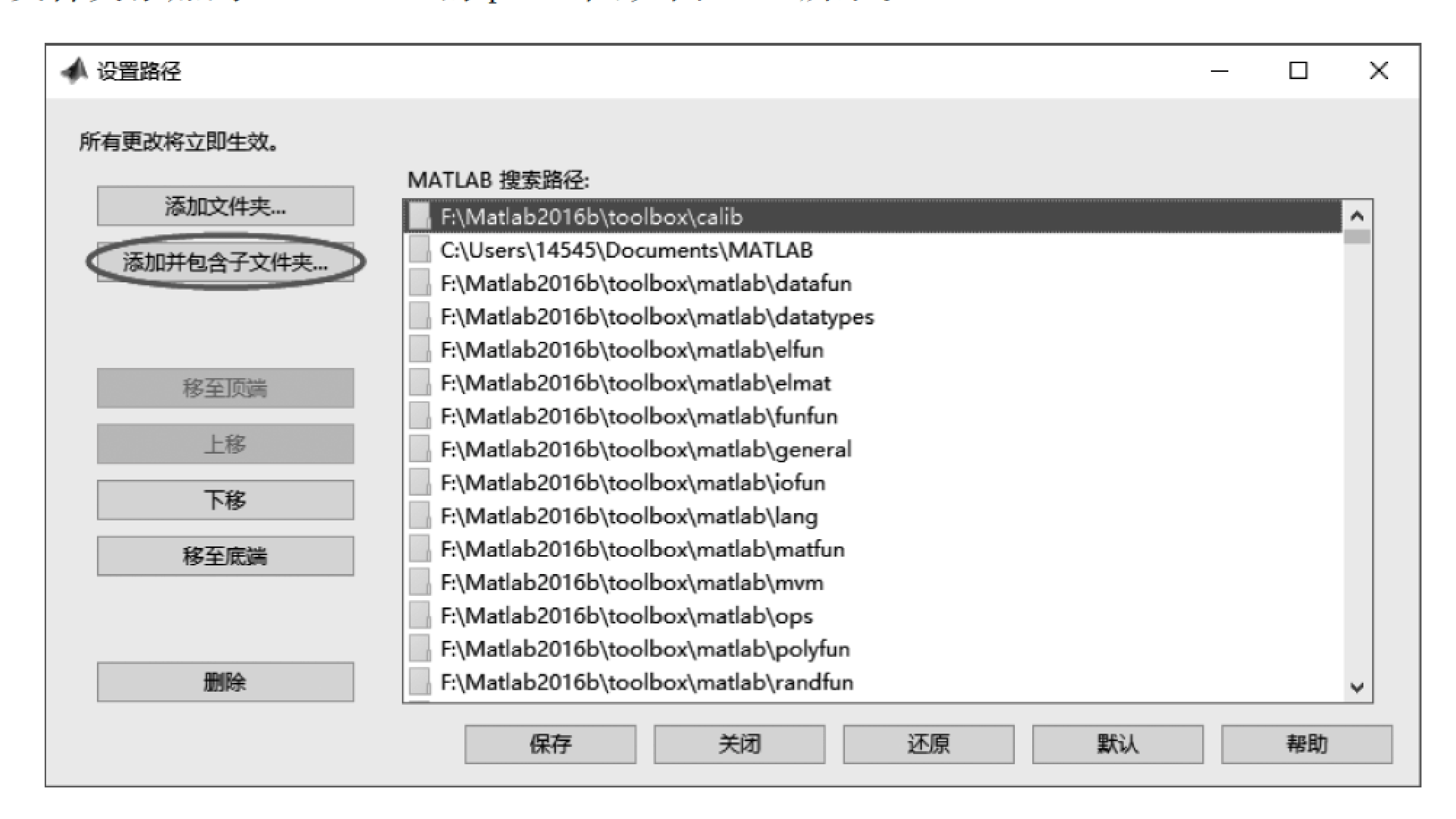


图 3-49　MATLAB 中添加图像路径第一步

将两个路径添加进去之后会在“设置路径”界面中出现两个路径，单击“保存”按钮之后关闭“设置路径”界面即可，如图 3-50 所示。

(5) 单个摄像头标定。

所有需要用到的路径添加完成后，在 MATLAB 命令窗口中输入 calib_gui，会出现如

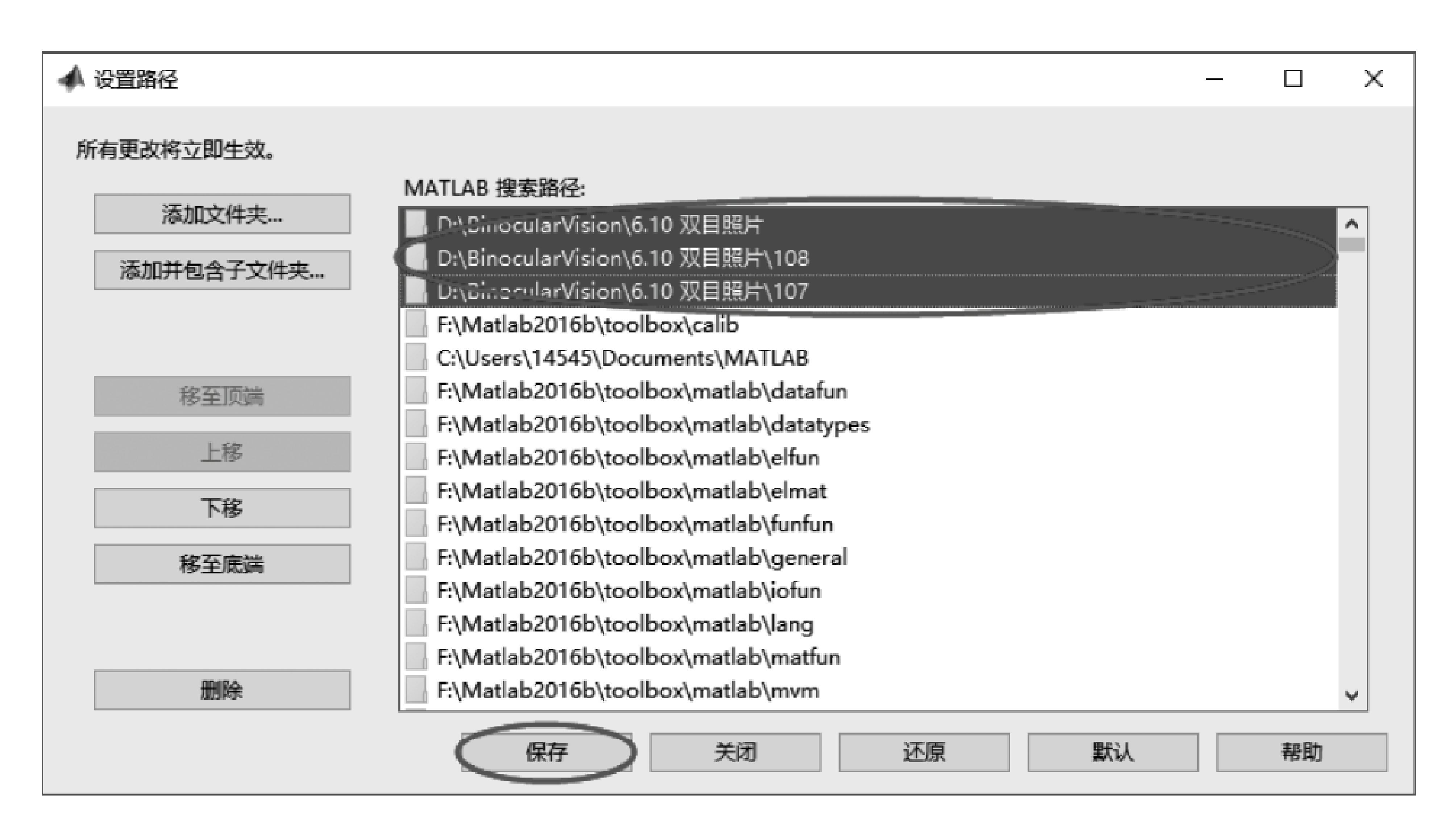


图 3-50　MATLAB 中添加图像路径第二步

图 3-51 所示的窗口，单击第一个选项 Standard(all the images are stored in memory)，会出现如图 3-52 所示的菜单栏。

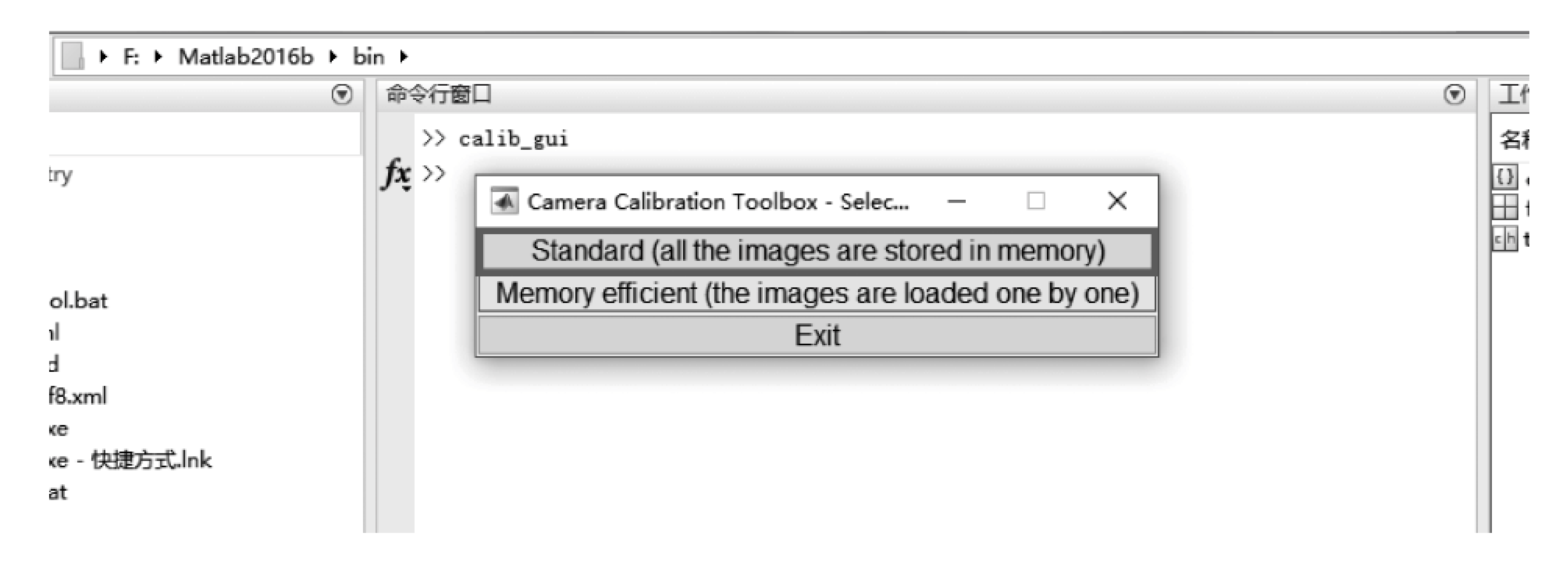


图 3-51　调用标定函数

Camera Calibration Toolbox - Standard Version

Image names	Read images	Extract grid corners	Calibration
Show Extrinsic	Reproject on images	Analyse error	Recomp. corners
Add/Suppress images	Save	Load	Exit
Comp. Extrinsic	Undistort image	Export calib data	Show calib results

图 3-52　标定菜单栏

回到 MATLAB 的命令行窗口中，将路径改为左图像存储的路径，如图 3-53 所示。

单击图 3-52 中的 Image names，进行图像读入。如图 3-54 所示为读入图像后的画面，图上方为文件夹中所有图像，在下方的第一行输入 left_表示从上述图像中仅提取文件名以 left 开始的图像。

在第二栏中输入 b 表示仅输入.bmp 格式的文件，导入成功后如图 3-55 和图 3-56 所示。

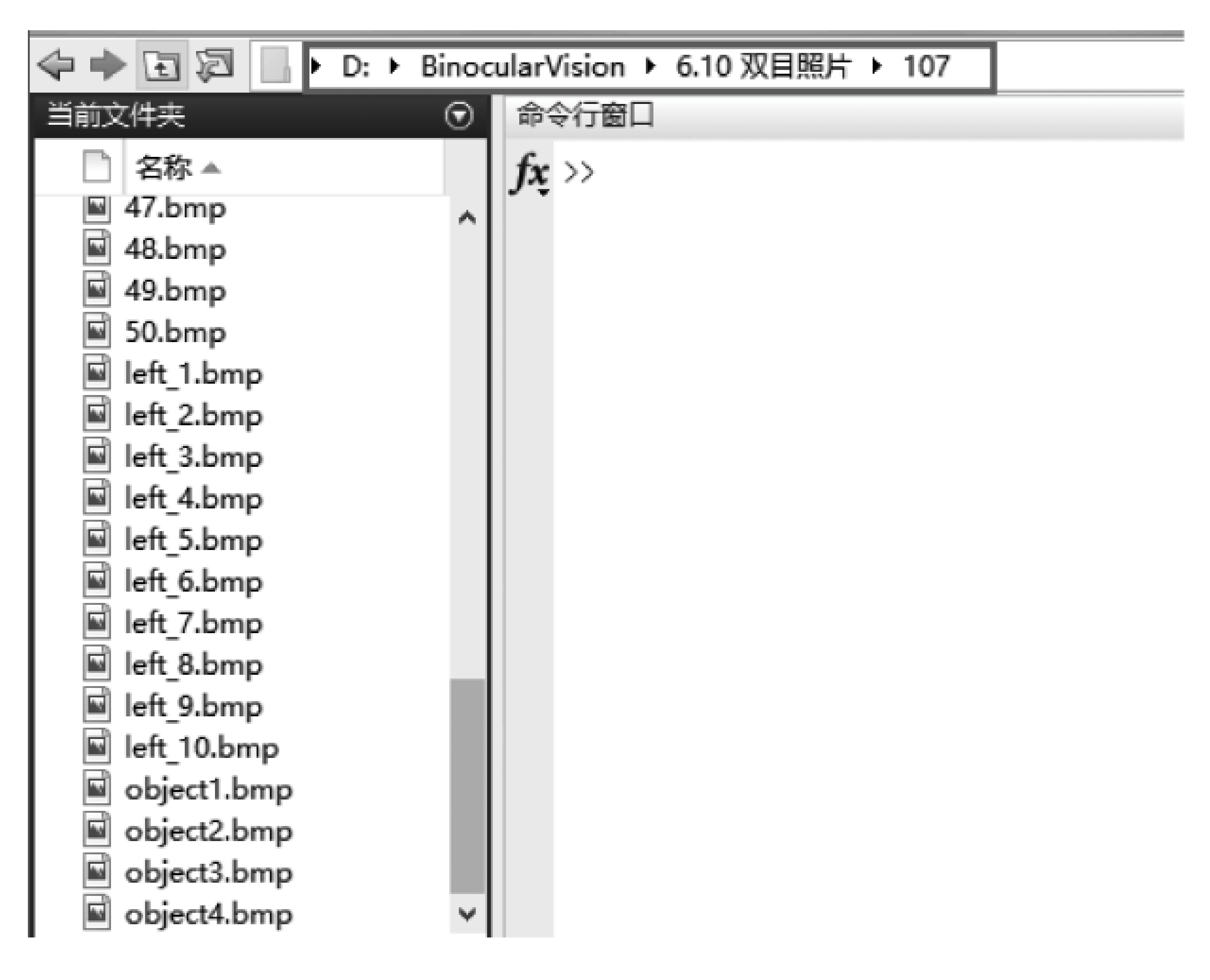


图 3-53 修改 Matlab 的路径

命令行窗口

```
>> calib_gui

.           19.bmp    29.bmp    39.bmp    49.bmp       left_8.bmp
..          20.bmp    30.bmp    40.bmp    50.bmp       left_9.bmp
11.bmp      21.bmp    31.bmp    41.bmp    left_1.bmp   object1.bmp
12.bmp      22.bmp    32.bmp    42.bmp    left_10.bmp  object2.bmp
13.bmp      23.bmp    33.bmp    43.bmp    left_2.bmp   object3.bmp
14.bmp      24.bmp    34.bmp    44.bmp    left_3.bmp   object4.bmp
15.bmp      25.bmp    35.bmp    45.bmp    left_4.bmp
16.bmp      26.bmp    36.bmp    46.bmp    left_5.bmp
17.bmp      27.bmp    37.bmp    47.bmp    left_6.bmp
18.bmp      28.bmp    38.bmp    48.bmp    left_7.bmp

fx Basename camera calibration images (without number nor suffix): left_
```

图 3-54 标定函数图像读入

```
Basename camera calibration images (without number nor suffix): left_
Image format: ([]='r'='ras', 'b'='bmp', 't'='tif', 'p'='pgm', 'j'='jpg', 'm'='ppm') b
Loading image 1...2...3...4...5...6...7...8...9...10...
done
```

图 3-55 标定函数成功读入图像 1

回到主控界面，单击 Extract grid corners 提取每幅图像的角点。单击完成之后，命令行会出现如图 3-57 所示的提示。这几项都可以直接按 Enter 键跳过，它们表示以多大的窗口去搜索棋盘窗，较大的窗口会更方便提取，即便点有适当偏离也能找到。

在按 Enter 键之后会跳出第一个棋盘窗，按照顺时针的顺序，依次选中棋盘中外侧第二圈的角点（不要单击最外侧的角点），如图 3-58 所示。

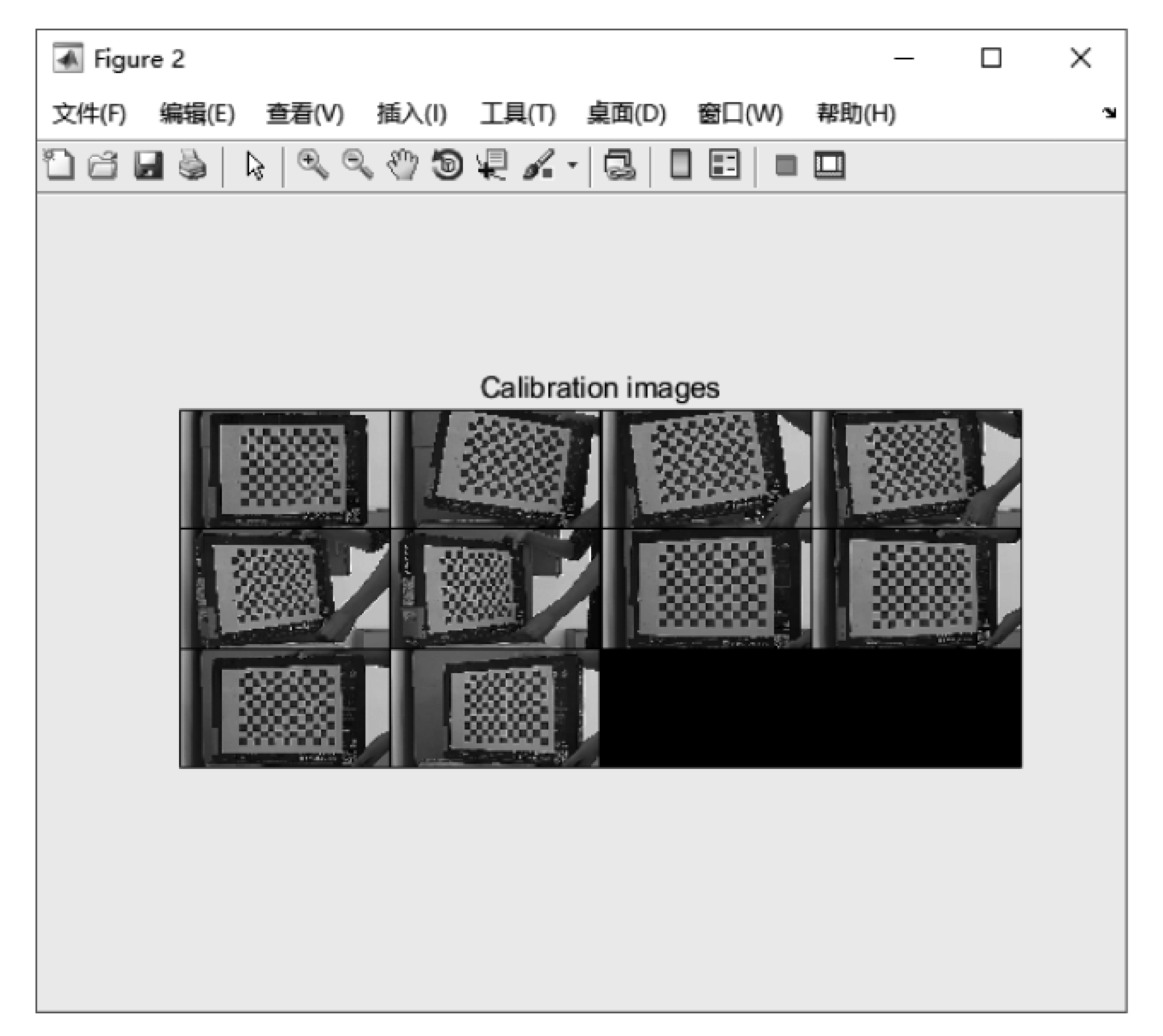


图 3-56　标定函数成功读入图像 2

```
Extraction of the grid corners on the images
Number(s) of image(s) to process ([] = all images) =
Window size for corner finder (wintx and winty):
wintx ([] = 10) =
winty ([] = 10) =
Window size = 21x21
Do you want to use the automatic square counting mechanism (0=[]=default)
  or do you always want to enter the number of squares manually (1,other)? 1
```

图 3-57　角点提取

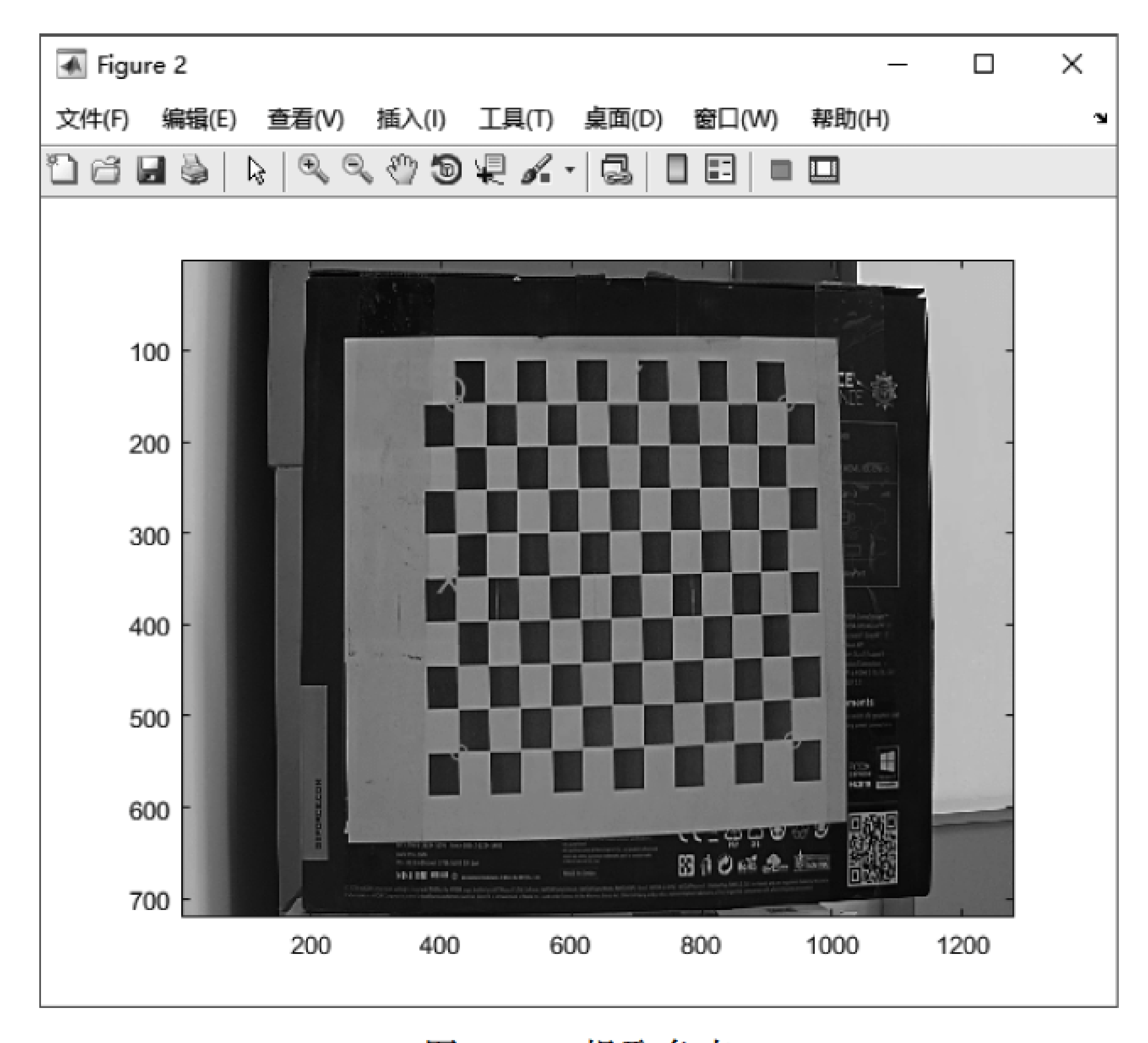


图 3-58　提取角点

单击之后需要输入一些棋盘的参数信息，首先输入图像中每一个方块的长和宽，单位是毫米，如图 3-59 所示。

```
Processing image 1...
Using (wintx,winty)=(10,10) - Window size = 21x21      (Note: To reset the window size, run script clearwin)
Click on the four extreme corners of the rectangular complete pattern (the first clicked corner is the origin)...
Size dX of each square along the X direction ([]=100mm) = 18
Size dY of each square along the Y direction ([]=100mm) = 18
If the guessed grid corners (red crosses on the image) are not close to the actual corners,
it is necessary to enter an initial guess for the radial distortion factor kc (useful for subpixel detection)
fx Need of an initial guess for distortion? ([]=no, other=yes)
```

图 3-59　输入标定参数

接下来按 Enter 键，依次标定剩下 9 张图像的角点，最后回到菜单界面。在菜单界面单击 Calibration 按钮进行标定。

标定完成后，即可得到标定的结果，单击 Show Extrinsic 可视化标定的结果，如图 3-60 所示。

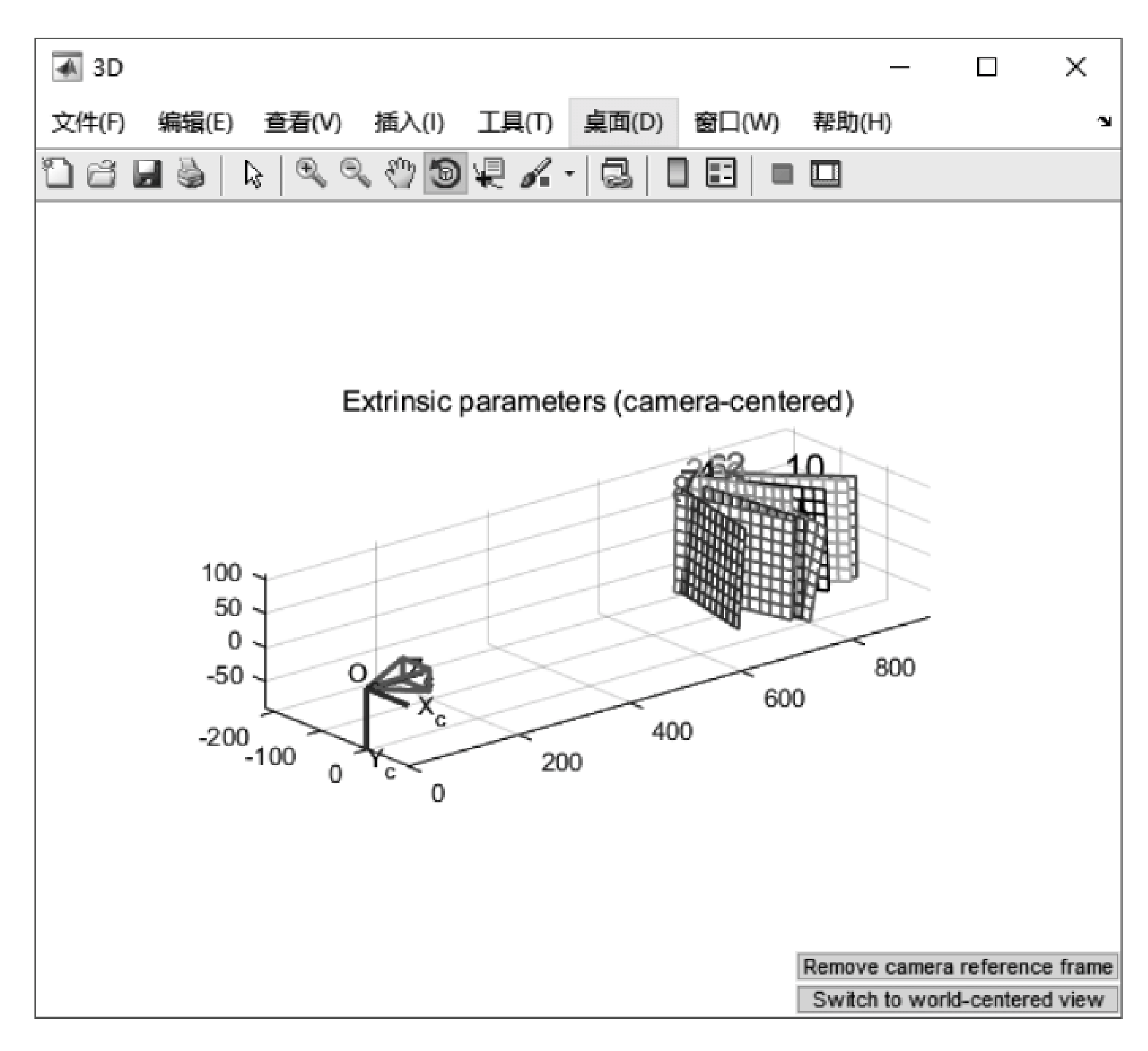


图 3-60　标定结果可视化

单击 Analyse Error 可以查看标定的误差情况，如图 3-61 所示。

标定过一个摄像头后，单击 Save 按钮保存结果。将保存的 Calib_Results. mat 文件的名称改为 Left_ Calib_Results. mat，并放在另一个文件夹中。

完成左摄像头标定后，对右摄像头的图像使用同样的方法进行标定，将保存的结果 Calib_Results. mat 名称改为 Right_Calib_Results. mat 并放在之前存放 Left_ Calib_Results. mat 的文件夹中。

左、右摄像头都标定完成后，在 MATLAB 的命令行窗口中输入 stereo_gui 启动立体标定板，如图 3-62 所示。

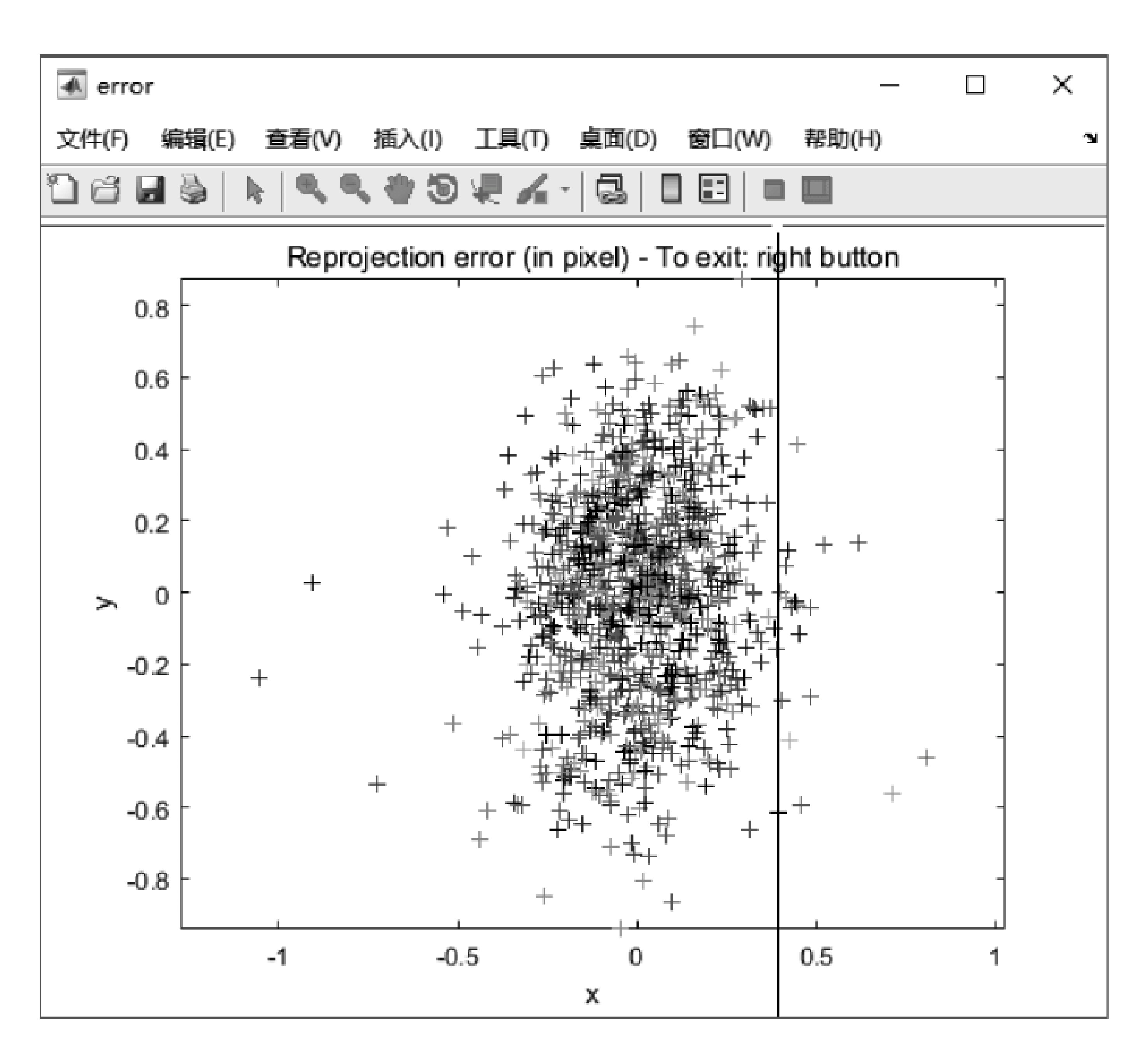


图 3-61　标定误差分析

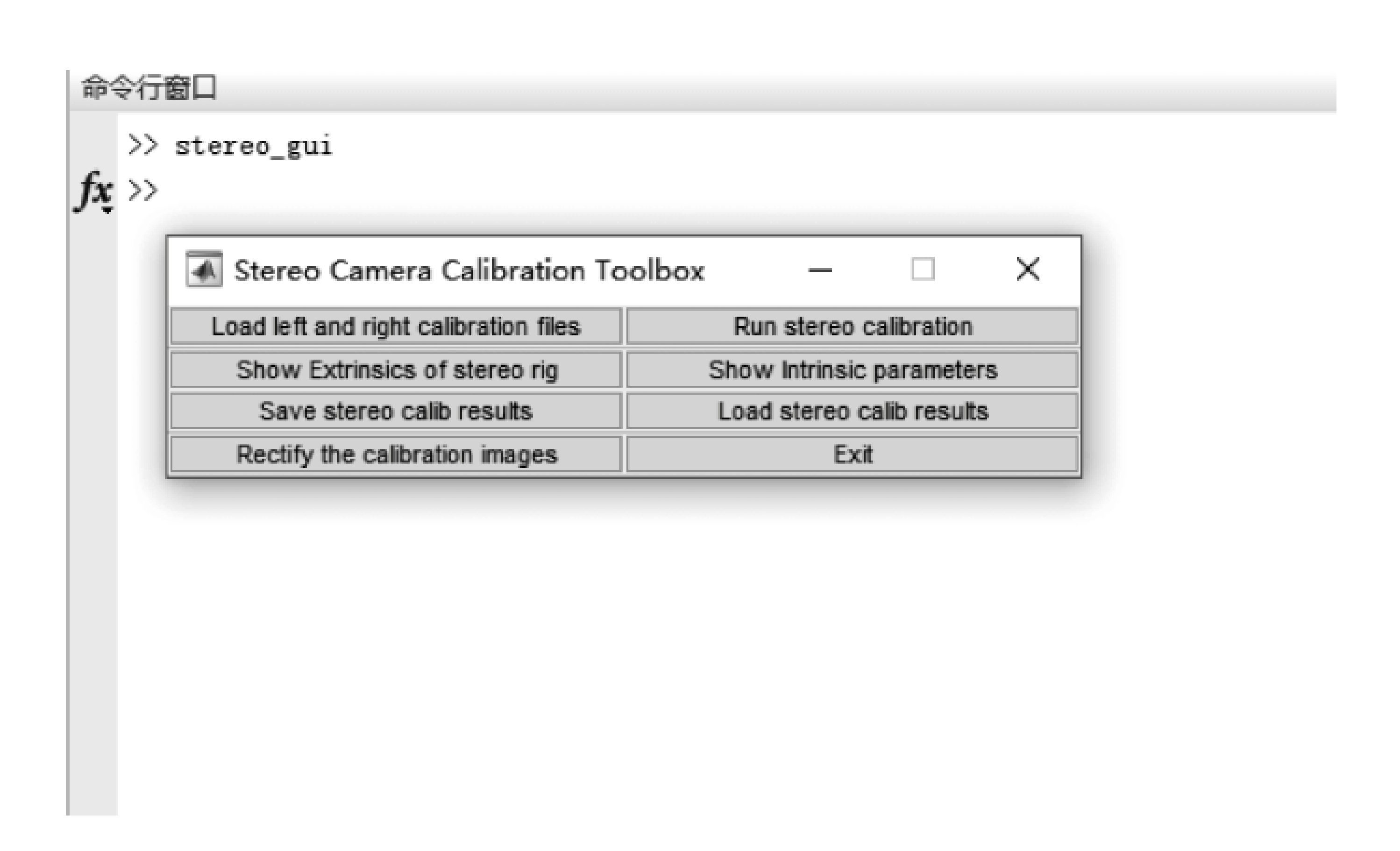


图 3-62　启动立体标定板

将 MATLAB 的默认路径改为存放两个.mat 文件的文件夹，如图 3-63 所示。

单击 Load left and right calibration files，在命令行窗口中分别填写左、右标定好的结果，如图 3-64 所示。

输入完成之后单击 Run stereo calibration 对左右参数进行修正，之后再单击 Show Extrinsics of stereo rig 即可将结果可视化，如图 3-65 和图 3-66 所示。

最后单击 Save stereo calib results 即可保存标定后的结果。

方案二：扩展标定 APP

(1) 在版本比较新的 MATLAB 中，可以在一侧找到附加的 APP，如图 3-67 所示，可以在其中找到一个名为 Stereo Camera Calibrator 的扩展 APP，其功能就是双目立体视觉的标定。

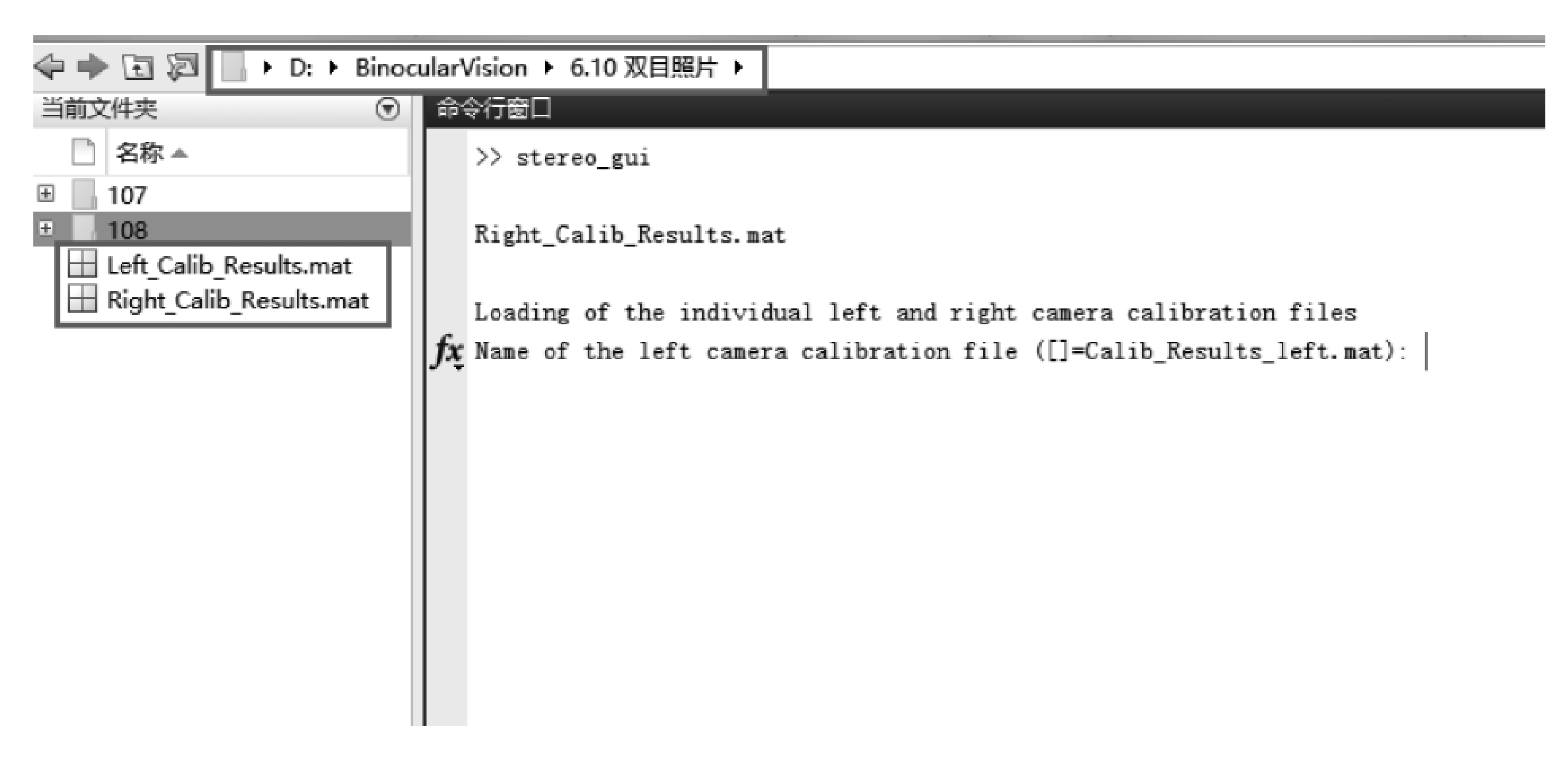


图 3-63　立体标定_修改路径

```
Loading of the individual left and right camera calibration files
Name of the left camera calibration file ([]=Calib_Results_left.mat): Left_Calib_Results.mat
Name of the right camera calibration file ([]=Calib_Results_right.mat): Right_Calib_Results.mat
Loading the left camera calibration result file Left_Calib_Results.mat...
Loading the right camera calibration result file Right_Calib_Results.mat...
```

图 3-64　输入两个.mat 文件

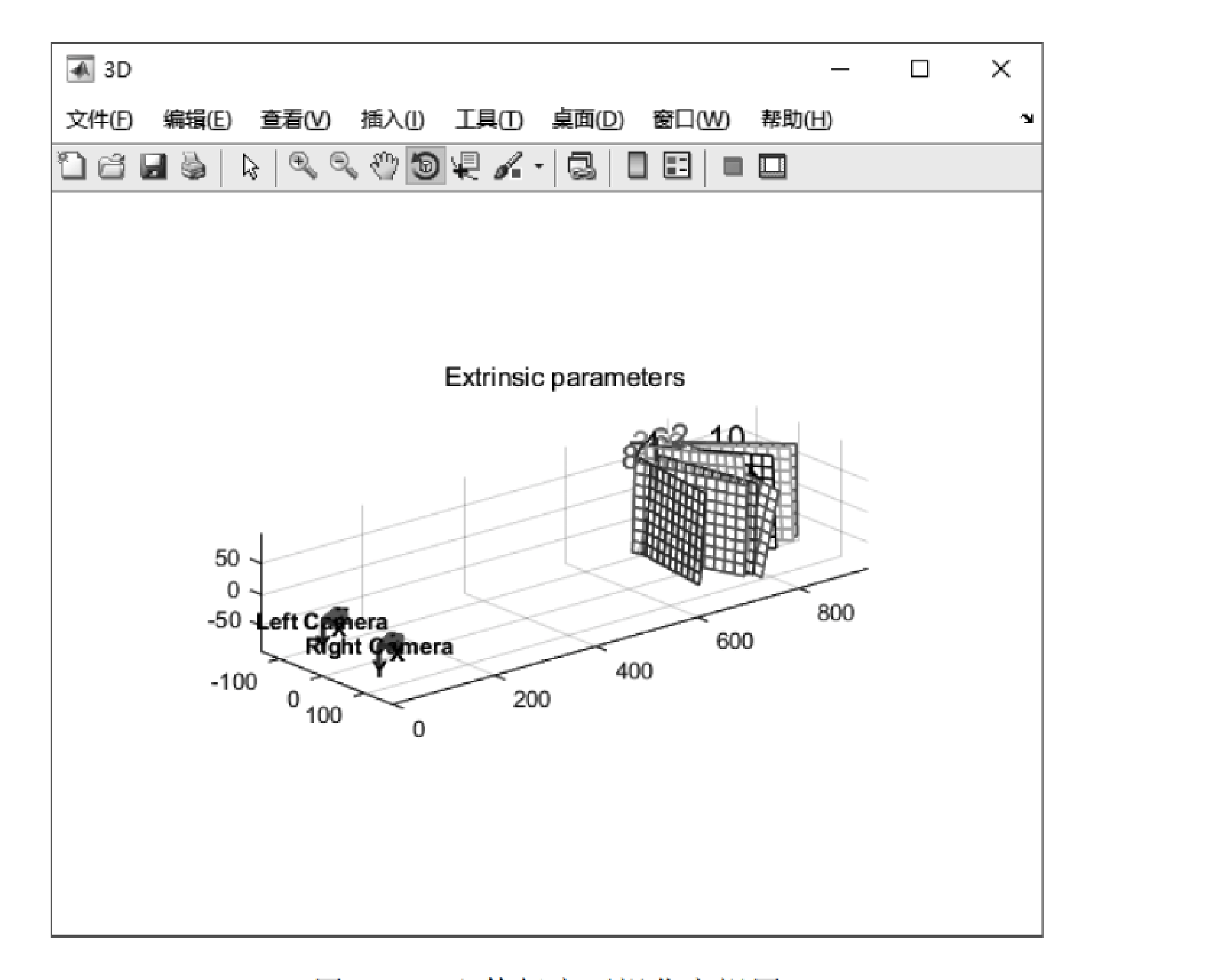


图 3-65　立体标定可视化主视图

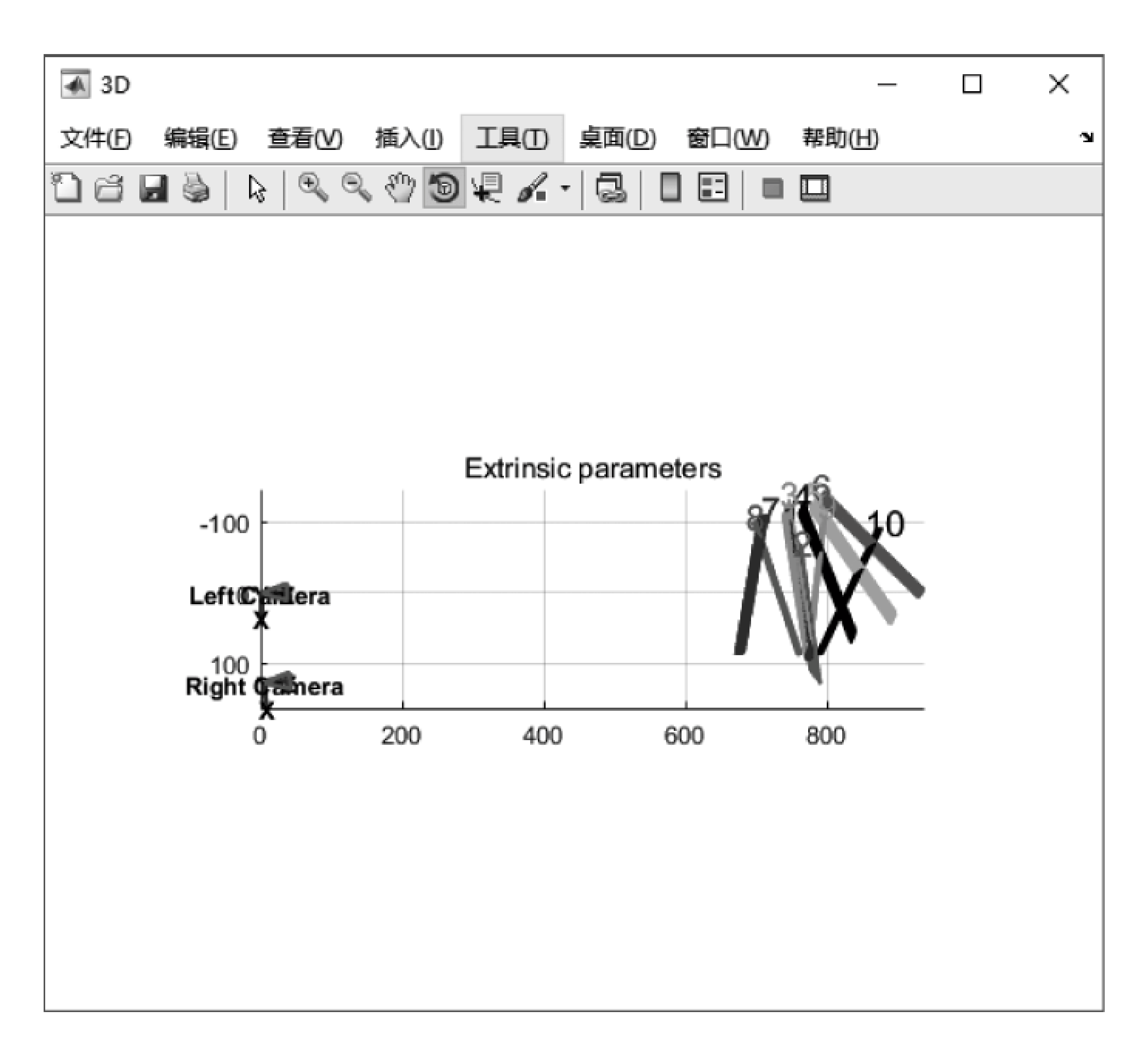


图 3-66 立体标定可视化俯视图

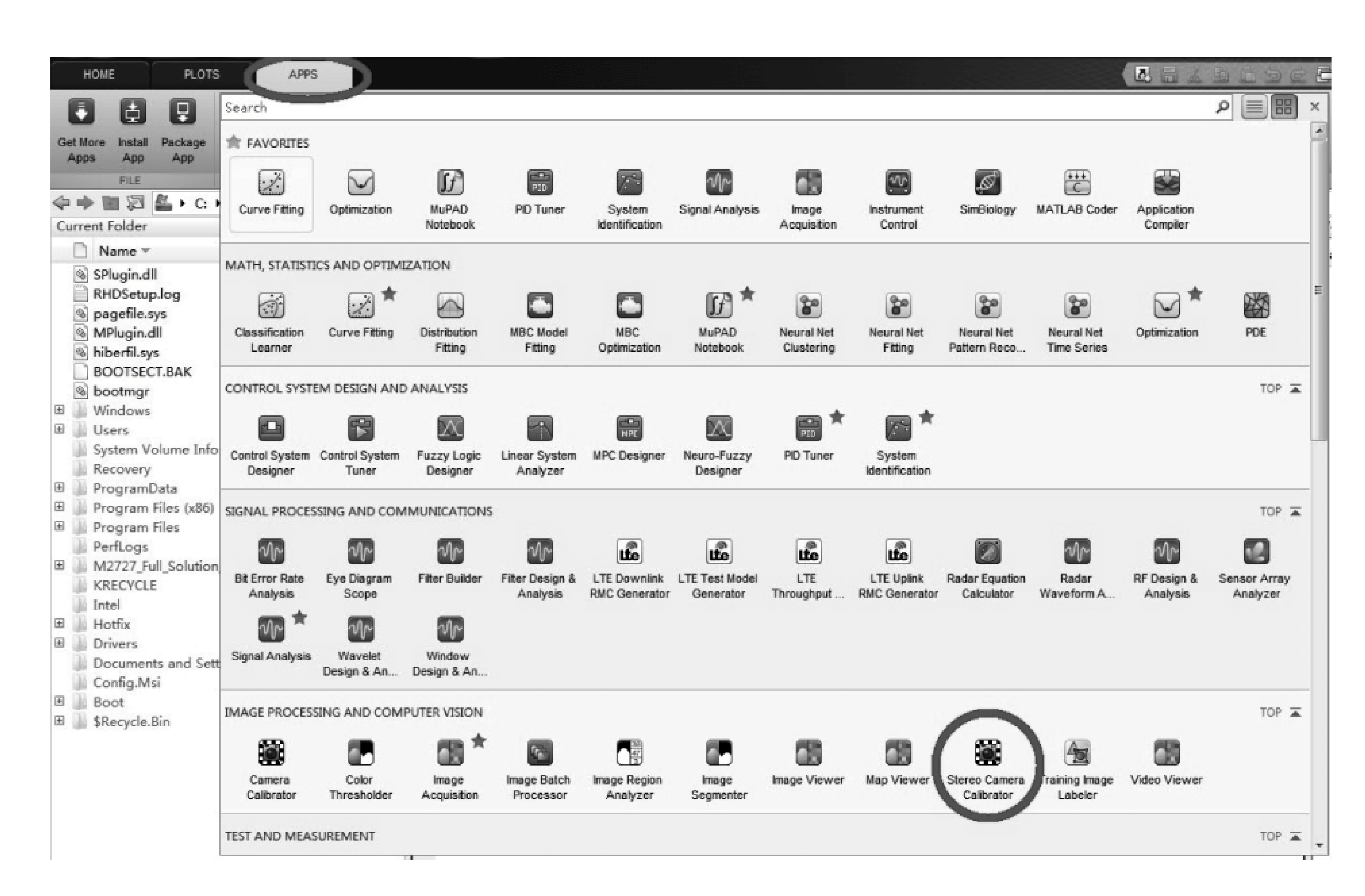


图 3-67 MATLAB 标定第一步

(2) 安装好 APP 后,如图 3-68 所示,单击左上角的 Add Images 添加左、右摄像头的图像。

(3) 在添加图片时,需要在上边的一栏中添加用于存储左摄像头拍摄到的图片的文件夹,下边则需要添加用于存储右摄像头拍摄到的图片的文件夹,如图 3-69 所示,最下边需要填好每一个标定格的宽度。

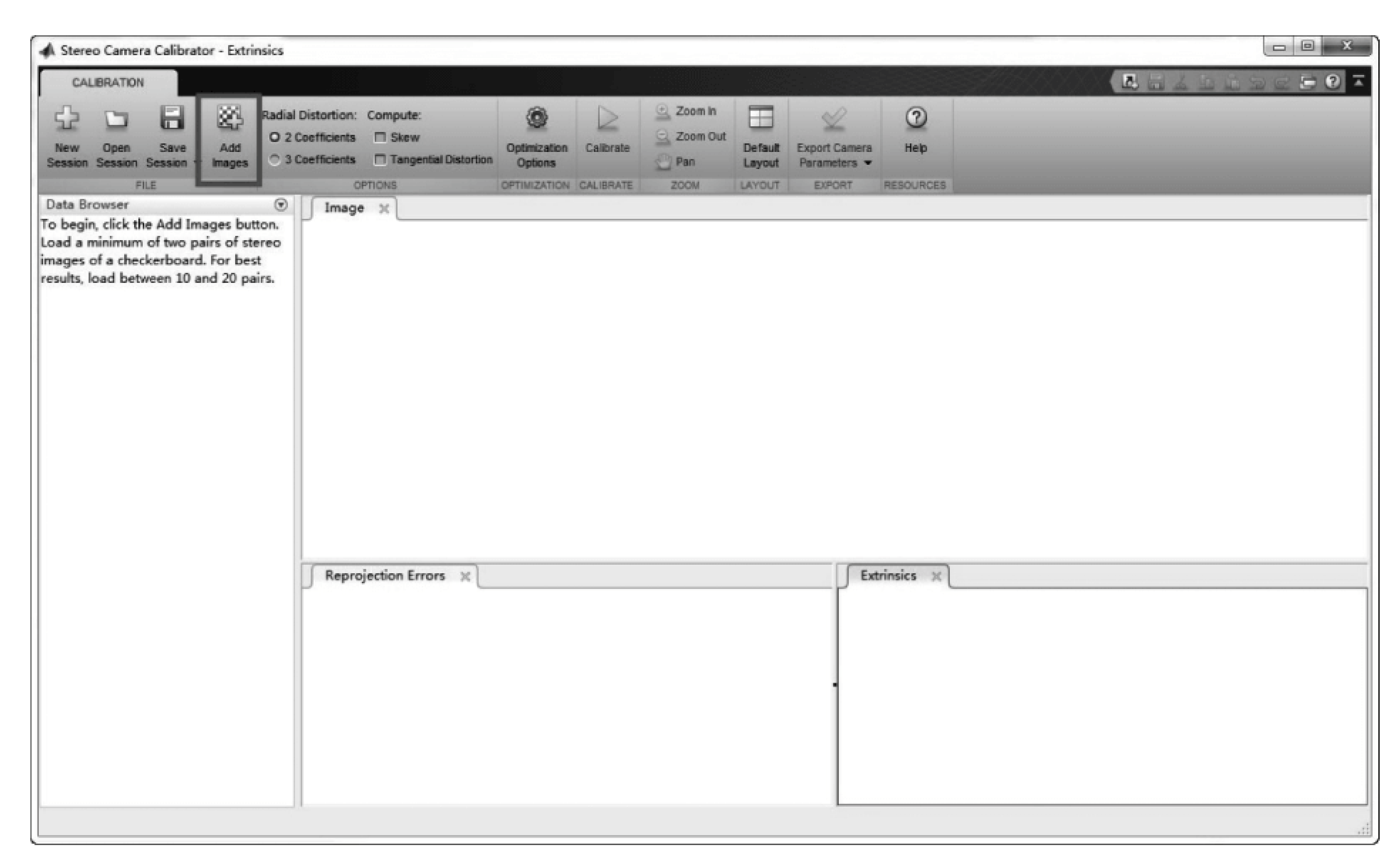


图 3-68　MATLAB 标定第二步

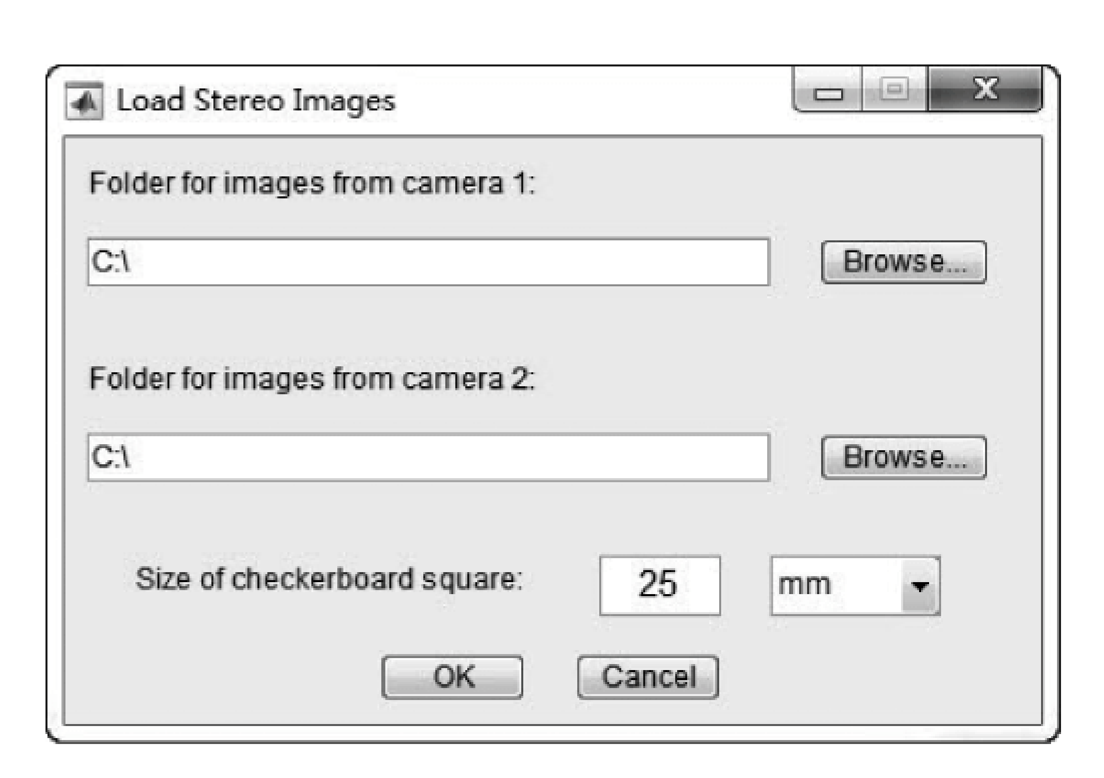


图 3-69　MATLAB 标定第三步

需要注意的是，虽然对文件夹中的图片名称没有明确要求，但是为了避免不必要的麻烦，提高匹配的成功率，一定要把拍摄到的图片按照对应的顺序排好，即按照标定板每个角度拍摄的顺序进行排序，如图 3-70 所示为示例文件夹存储的图片。

(4) 之后进入标定，如图 3-71 所示，在这个过程中耐心等待即可。

标定完成后如图 3-72 所示，可以删除标定效果不好的图片，以提高标定的精度。本示例中使用的图片较少，在实际标定过程中使用的图片越多越好。

(5) 在完成整个标定流程后，可以单击"保存"按钮存储并关闭，将摄像机的参数以 MATLAB 特有的. mat 文件进行存储，为后续测深度提供摄像头参数，如图 3-73 所示。

3.5.3　OpenCV 实现

在本示例的测量物体深度程序中，采用了 OpenCV 进行摄像机标定。因为使用的是双目摄像头，所以在读入时需要先做简单的图像分割，之后再进行 SIFT 特征提取匹配生成视差图，最后测得物体深度。



图 3-70　MATLAB 标定第四步

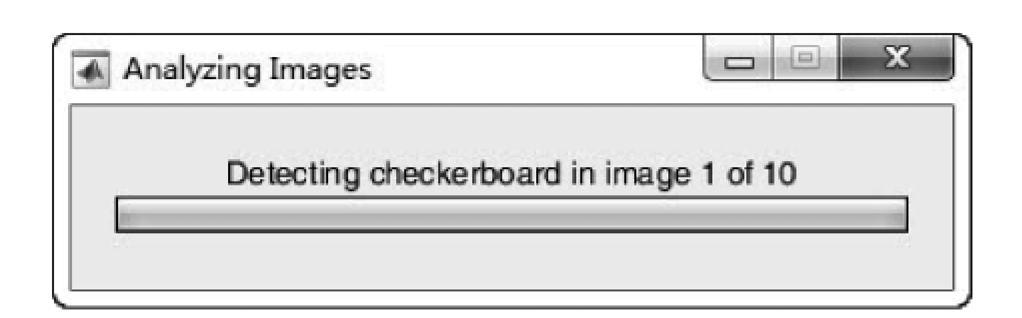


图 3-71　MATLAB 标定第五步

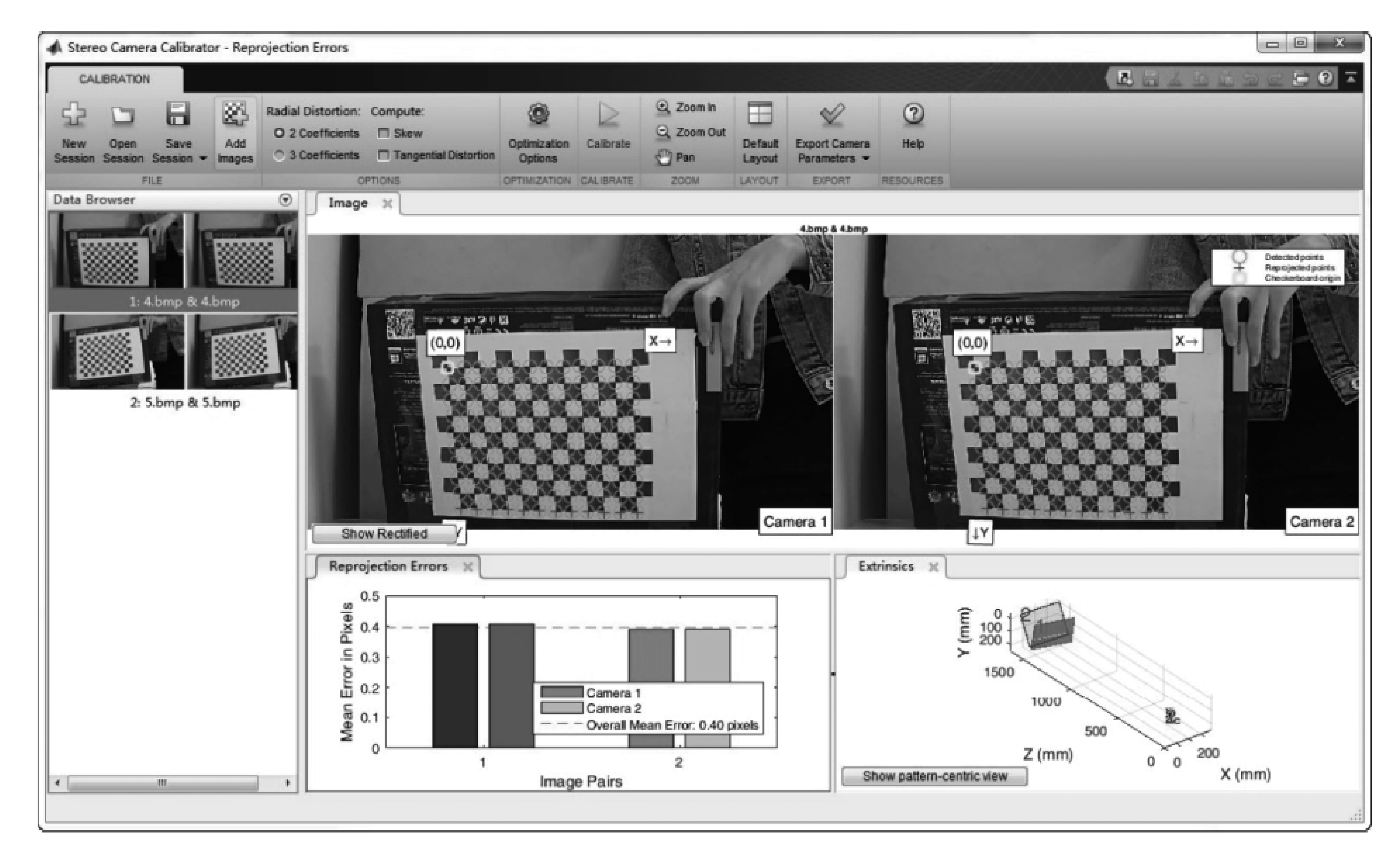


图 3-72　实际标定结果

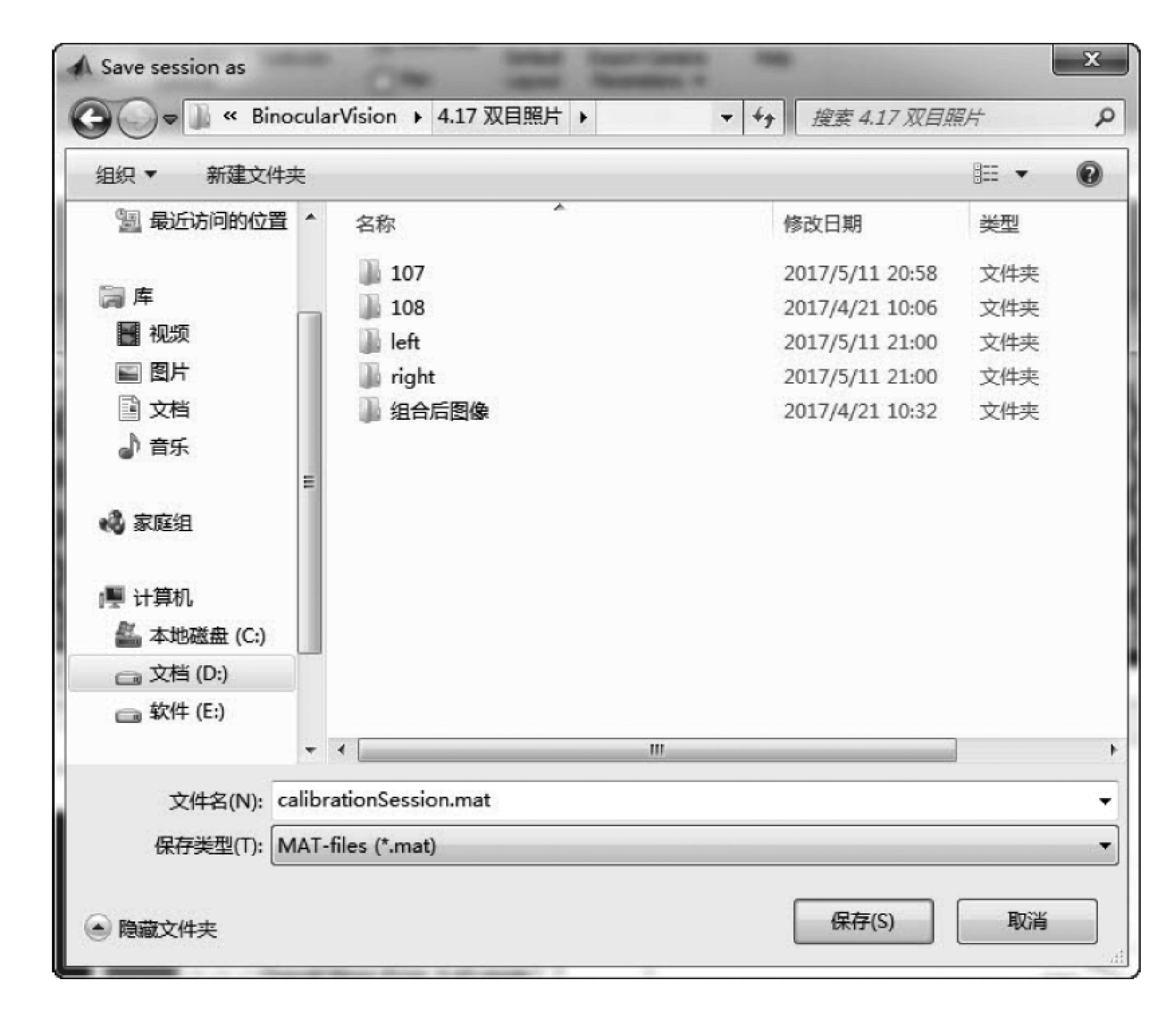


图 3-73　存储为.mat 文件

1. 程序读入的图像

(1) 程序运行前需要输入多组标定图像和实际场景的图像,如图 3-74 所示为其中一组标定图像。

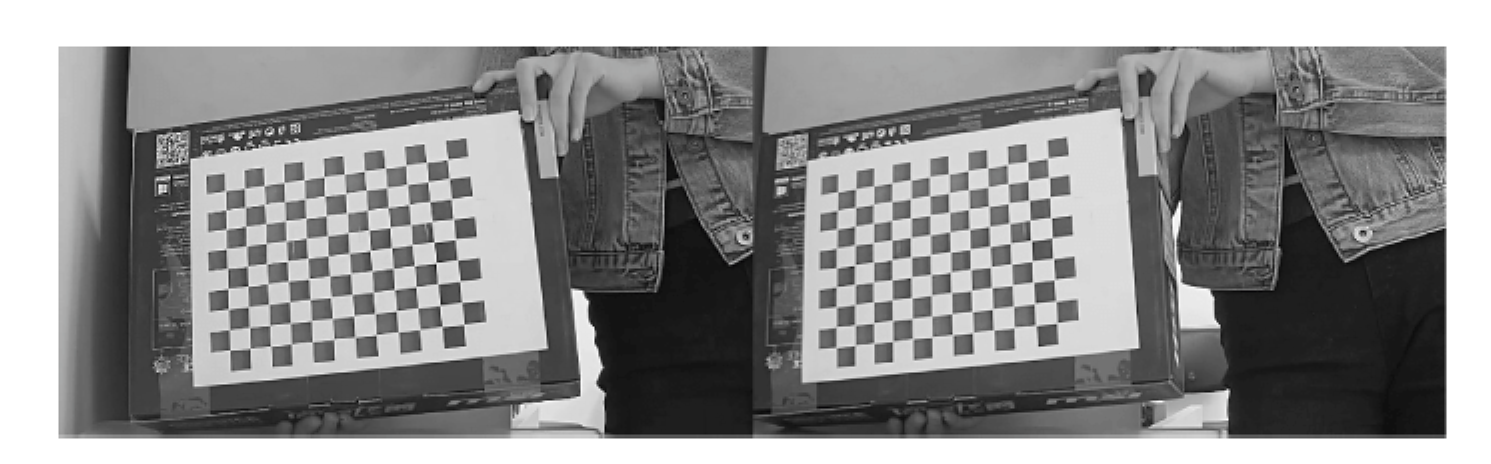


图 3-74　OpenCV 标定

(2) 如图 3-75 所示为需要测定深度的图像。



图 3-75　被测图像

2. 程序实现

```
#include <opencv2/opencv.hpp>
#include <opencv2\imgproc\imgproc.hpp>
#include <opencv2\core\core.hpp>
#include <opencv2\highgui\highgui.hpp>
#include <opencv2\calib3d\calib3d.hpp>
#include <opencv2\features2d\features2d.hpp>
#include <opencv2\legacy\legacy.hpp>
#include <iostream>
#include <fstream>
#include <vector>
#include <list>
#include <algorithm>
#include <iterator>
#include <cstdio>
#include <string>

using namespace cv;
using namespace std;

typedef unsigned int uint;

Size imgSize(1280, 720);
Size patSize(12,9);                       //内半圈的脚点个数 12*9
const double patLen = 18.0f;              //unit: mm 标定板每个格的宽度(金属标定板)
double imgScale = 1.0;                    //图像缩放的比例因子


//将要读取的图片路径存储在 fileList 中
vector<string> fileList;
void initFileList(string dir, int first, int last){
    fileList.clear();
    for(int cur = first; cur <= last; cur++){
        string str_file = dir + "/" + to_string(cur) + ".jpg";
        fileList.push_back(str_file);
    }
}


//生成点云坐标后保存
static void saveXYZ(string filename, const Mat& mat)
{
     const double max_z = 1.0e4;
     ofstream fp(filename);
    if (!fp.is_open())
     {
           std::cout <<"打开点云文件失败"<< endl;
fp.close();
        return;
     }
```

```
    //遍历写入
for(int y = 0; y < mat.rows; y++)
     {
for(int x = 0; x < mat.cols; x++)
        {
            Vec3f point = mat.at < Vec3f >(y, x);     //三通道浮点型
            if(fabs(point[2] - max_z) < FLT_EPSILON || fabs(point[2]) > max_z)
              continue;
            fp << point[0]<<" "<< point[1]<<" "<< point[2]<< endl;
        }
     }
    fp.close();
}


//存储视差数据
void saveDisp(const string filename, const Mat& mat)
{
    ofstream fp(filename, ios::out);
    fp << mat.rows << endl;
    fp << mat.cols << endl;
    for(int y = 0; y < mat.rows; y++)
    {
        for(int x = 0; x < mat.cols; x++)
        {
            double disp = mat.at < short >(y, x);     //这里视差矩阵是 CV_16S 格式的,故用
                                                        short 类型读取
            fp << disp << endl;                        //若视差矩阵是 CV_32F 格式,则用 float
                                                        类型读取
        }
    }
    fp.close();
}


void F_Gray2Color(Mat gray_mat, Mat& color_mat)
{
    color_mat = Mat::zeros(gray_mat.size(), CV_8UC3);
    int rows = color_mat.rows, cols = color_mat.cols;

    Mat red = Mat(gray_mat.rows, gray_mat.cols, CV_8U);
    Mat green = Mat(gray_mat.rows, gray_mat.cols, CV_8U);
    Mat blue = Mat(gray_mat.rows, gray_mat.cols, CV_8U);
    Mat mask = Mat(gray_mat.rows, gray_mat.cols, CV_8U);

    subtract(gray_mat, Scalar(255), blue);   //blue(I) = 255 - gray(I)
    red = gray_mat.clone();                  //red(I) = gray(I)
    green = gray_mat.clone();                //green(I) = gray(I),if gray(I) < 128

    compare(green, 128, mask, CMP_GE);       //green(I) = 255 - gray(I), if gray(I) >= 128
    subtract(green, Scalar(255), green, mask);
```

```
        convertScaleAbs(green, green, 2.0, 2.0);

        vector<Mat> vec;
        vec.push_back(red);
        vec.push_back(green);
        vec.push_back(blue);
        cv::merge(vec, color_mat);
    }

    Mat F_mergeImg(Mat img1, Mat disp8){
        Mat color_mat = Mat::zeros(img1.size(), CV_8UC3);

        Mat red = img1.clone();
        Mat green = disp8.clone();
        Mat blue = Mat::zeros(img1.size(), CV_8UC1);

        vector<Mat> vec;
        vec.push_back(red);
        vec.push_back(blue);
        vec.push_back(green);
        cv::merge(vec, color_mat);

        return color_mat;
    }


    //双目立体标定
    int stereoCalibrate(string intrinsic_filename = "intrinsics.yml", string extrinsic_filename =
    "extrinsics.yml")
    {
        vector<int> idx;
        //左侧相机的角点坐标和右侧相机的角点坐标

        vector<vector<Point2f>> imagePoints[2];

        //vector<vector<Point2f>> leftPtsList(fileList.size());
        //vector<vector<Point2f>> rightPtsList(fileList.size());

        for(uint i = 0; i < fileList.size();++i)
        {
            vector<Point2f> leftPts, rightPts;     //存储左、右相机的角点位置
            Mat rawImg = imread(fileList[i]);       //原始图像
            if(rawImg.empty()){
                std::cout <<"the Image is empty..."<< fileList[i]<< endl;
                continue;
            }
            //截取左右图片
            Rect leftRect(0, 0, imgSize.width, imgSize.height);
            Rect rightRect(imgSize.width, 0, imgSize.width, imgSize.height);

            Mat leftRawImg = rawImg(leftRect);      //切分得到的左原始图像
```

```
        Mat rightRawImg = rawImg(rightRect);    //切分得到的右原始图像

        imwrite("left.jpg", leftRawImg);
        imwrite("right.jpg", rightRawImg);
        //std::cout <<"左侧图像:宽度"<< leftRawImg.size().width <<" 高度"<< rightRawImg.
size().height << endl;
        //std::cout <<"右侧图像:宽度"<< rightRawImg.size().width <<" 高度"<< rightRawImg.
size().height << endl;

        Mat leftImg, rightImg, leftSimg, rightSimg, leftCimg, rightCimg, leftMask, rightMask;
        //BGT -> GRAY
        if(leftRawImg.type() == CV_8UC3)
            cvtColor(leftRawImg, leftImg, CV_BGR2GRAY);  //转为灰度图
         else
           leftImg = leftRawImg.clone();
        if(rightRawImg.type() == CV_8UC3)
            cvtColor(rightRawImg, rightImg, CV_BGR2GRAY);
        else
            rightImg = rightRawImg.clone();

        imgSize = leftImg.size();

        //图像滤波预处理
        resize(leftImg, leftMask, Size(200, 200));        //resize 对原图像 img 重新调整大小
                                                           生成 mask 图像大小为 200 * 200
        resize(rightImg, rightMask, Size(200, 200));
GaussianBlur(leftMask, leftMask, Size(13, 13), 7);
        GaussianBlur(rightMask, rightMask, Size(13, 13), 7);
        resize(leftMask, leftMask, imgSize);
        resize(rightMask, rightMask, imgSize);
          medianBlur(leftMask, leftMask, 9);               //中值滤波
        medianBlur(rightMask, rightMask, 9);

          for (int v = 0; v < imgSize.height; v++) {
              for (int u = 0; u < imgSize.width; u++) {
                  int leftX = ((int)leftImg.at<uchar>(v, u) - (int)leftMask.at
<uchar>(v, u)) * 2 + 128;
                int rightX = ((int)rightImg.at<uchar>(v, u) - (int)rightMask.at
<uchar>(v, u)) * 2 + 128;
                leftImg.at<uchar>(v, u) = max(min(leftX, 255), 0);
                rightImg.at<uchar>(v, u) = max(min(rightX, 255), 0);
              }
          }

        //寻找角点,图像缩放
        resize(leftImg, leftSimg, Size(), imgScale, imgScale);
                                                    //图像以 0.5 的比例缩放
        resize(rightImg, rightSimg, Size(), imgScale, imgScale);
          cvtColor(leftSimg, leftCimg, CV_GRAY2BGR);
                                                    //转为 BGR 图像,cimg 和 simg 是 800×600 的图像
        cvtColor(rightSimg, rightCimg, CV_GRAY2BGR);
```

```
        //寻找棋盘角点
        bool leftFound = findChessboardCorners(leftCimg, patSize, leftPts, CV_CALIB_CB_
ADAPTIVE_THRESH|CV_CALIB_CB_FILTER_QUADS);
        bool rightFound = findChessboardCorners(rightCimg, patSize, rightPts, CV_CALIB_CB_
ADAPTIVE_THRESH|CV_CALIB_CB_FILTER_QUADS);

        if(leftFound)
            cornerSubPix(leftSimg, leftPts, Size(11, 11), Size( - 1, - 1 ),
                TermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 300, 0.01));
        if(rightFound)
            cornerSubPix(rightSimg, rightPts, Size(11, 11), Size( - 1, - 1),
                TermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 300, 0.01));
                                             //亚像素

        //放大为原来的尺度
        for(uint j = 0;j < leftPts.size();j++)          //该幅图像共 132 个角点,坐标乘以 2,
                                                          还原角点位置
            leftPts[j] * = 1./imgScale;
        for(uint j = 0;j < rightPts.size();j++)
            rightPts[j] * = 1./imgScale;

        //显示
        string leftWindowName = "Left Corner Pic", rightWindowName = "Right Corner Pic";

        Mat leftPtsTmp = Mat(leftPts) * imgScale;      //再次乘以 imgScale
        Mat rightPtsTmp = Mat(rightPts) * imgScale;

        drawChessboardCorners(leftCimg, patSize, leftPtsTmp, leftFound);
                                             //绘制角点坐标并显示
        imshow(leftWindowName, leftCimg);
        imwrite("输出/DrawChessBoard/" + to_string(i) + "_left.jpg", leftCimg);
        waitKey(200);

        drawChessboardCorners(rightCimg, patSize, rightPtsTmp, rightFound);
                                             //绘制角点坐标并显示
        imshow(rightWindowName, rightCimg);
        imwrite("输出/DrawChessBoard/" + to_string(i) + "_right.jpg", rightCimg);
        waitKey(200);

        cv::destroyAllWindows();

        //保存角点坐标
        if(leftFound && rightFound)
        {
            imagePoints[0].push_back(leftPts);
            imagePoints[1].push_back(rightPts);          //保存角点坐标
            std::cout <<"图片 "<< i <<" 处理成功!"<< endl;
            idx.push_back(i);
        }
    }
    cv::destroyAllWindows();
```

```
    imagePoints[0].resize(idx.size());
    imagePoints[1].resize(idx.size());
    std::cout <<"成功标定的标定板个数为"<< idx.size()<<" 序号分别为: ";
    for(unsigned int i = 0;i < idx.size();++i)
        std::cout << idx[i]<<" ";

    //生成物点坐标
    vector < vector < Point3f >> objPts(idx.size());      //idx.size 代表成功检测的图像的个数
    for (int y = 0; y < patSize.height; y++) {
        for (int x = 0; x < patSize.width; x++) {
            objPts[0].push_back(Point3f((float)x, (float)y, 0) * patLen);
        }
    }
    for (uint i = 1; i < objPts.size(); i++) {
        objPts[i] = objPts[0];
    }

    //双目立体标定
    Mat cameraMatrix[2], distCoeffs[2];
    vector < Mat > rvecs[2], tvecs[2];
cameraMatrix[0] = Mat::eye(3, 3, CV_64F);
cameraMatrix[1] = Mat::eye(3, 3, CV_64F);
    Mat R, T, E, F;

    cv::calibrateCamera(objPts, imagePoints[0], imgSize, cameraMatrix[0],
distCoeffs[0], rvecs[0], tvecs[0], CV_CALIB_FIX_K3);

    cv::calibrateCamera(objPts, imagePoints[1], imgSize, cameraMatrix[1],
distCoeffs[1], rvecs[1], tvecs[1], CV_CALIB_FIX_K3);

    std::cout << endl <<"Left Camera Matrix: "<< endl << cameraMatrix[0]<< endl;
    std::cout << endl <<"Right Camera Matrix:"<< endl << cameraMatrix[1]<< endl;
    std::cout << endl <<"Left Camera DistCoeffs: "<< endl << distCoeffs[0]<< endl;
    std::cout << endl <<"Right Camera DistCoeffs: "<< endl << distCoeffs[1]<< endl;

    double rms = stereoCalibrate(objPts, imagePoints[0], imagePoints[1],
cameraMatrix[0], distCoeffs[0],
cameraMatrix[1], distCoeffs[1],
                      imgSize, R, T, E, F,
TermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 100, 1e-5));
                      //CV_CALIB_USE_INTRINSIC_GUESS);


    std::cout << endl << endl <<"立体标定完成! "<< endl <<"done with RMS error = " << rms << endl;
//反向投影误差
    std::cout << endl <<"Left Camera Matrix: "<< endl << cameraMatrix[0]<< endl;
    std::cout << endl <<"Right Camera Matrix:"<< endl << cameraMatrix[1]<< endl;
    std::cout << endl <<"Left Camera DistCoeffs: "<< endl << distCoeffs[0]<< endl;
    std::cout << endl <<"Right Camera DistCoeffs: "<< endl << distCoeffs[1]<< endl;
```

```
    //标定精度检测
    //通过检查图像上点与另一幅图像的极线的距离来评价标定的精度.为了实现这个目的,使用
undistortPoints 来对原始点做去畸变的处理
    //随后使用 computeCorrespondEpilines 来计算极线,计算点和线的点积.累计的绝对误差形成
了误差
    std::cout << endl <<" 极线计算... 误差计算... ";
    double err = 0;
     int npoints = 0;
     vector<Vec3f> lines[2];
for(unsigned int i = 0; i < idx.size(); i++)
     {
         int npt = (int)imagePoints[0][i].size();                  //角点个数
         Mat imgpt[2];
for(int k = 0; k < 2; k++)
         {
             imgpt[k] = Mat(imagePoints[k][i]);
              undistortPoints(imgpt[k], imgpt[k], cameraMatrix[k], distCoeffs[k], Mat(),
cameraMatrix[k]);                                                  //畸变
             computeCorrespondEpilines(imgpt[k], k+1, F, lines[k]);  //计算极线
         }
for(int j = 0; j < npt; j++)
         {
             double errij = fabs(imagePoints[0][i][j].x * lines[1][j][0] +
                                 imagePoints[0][i][j].y * lines[1][j][1] + lines[1][j][2]) +
                                 fabs(imagePoints[1][i][j].x * lines[0][j][0] +
                                 imagePoints[1][i][j].y * lines[0][j][1] + lines[0][j][2]);
             err += errij;                                         //累计误差
         }
         npoints += npt;
     }
     std::cout << " 平均误差 average reprojection err = " << err/npoints << endl;
                                                                   //平均误差

     //相机内参数和畸变系数写入文件
     FileStorage fs(intrinsic_filename, CV_STORAGE_WRITE);
     if(fs.isOpened())
     {
         fs << "M1" << cameraMatrix[0] << "D1" << distCoeffs[0] <<
             "M2" << cameraMatrix[1] << "D2" << distCoeffs[1];
fs.release();
     }
     else
         std::cout << "Error: can not save the intrinsic parameters\n";

    //立体矫正 BOUGUET'S METHOD
     Mat R1, R2, P1, P2, Q;
     Rect validRoi[2];

cv::stereoRectify(cameraMatrix[0], distCoeffs[0],
cameraMatrix[1], distCoeffs[1],
                     imgSize, R, T, R1, R2, P1, P2, Q,
```

```
                    CALIB_ZERO_DISPARITY, 1, imgSize, &validRoi[0], &validRoi[1]);

fs.open(extrinsic_filename, CV_STORAGE_WRITE);
if( fs.isOpened() )
    {
        fs << "R" << R << "T" << T << "R1" << R1 << "R2" << R2 << "P1" << P1 << "P2" << P2 << "Q" << Q;
fs.release();
    }
    else
        std::cout << "Error: can not save the intrinsic parameters\n";

    std::cout <<"双目标定完成..."<< endl;
    printf("\n输入任意字母进行下一步\n");
    getchar();getchar();
    return 0;
}


//-----------------------------------------------------------
//-----------------------------------------------------------
//双目立体匹配和测量
int stereoMatch(int picNum,
                string intrinsic_filename = "intrinsics.yml",
                string extrinsic_filename = "extrinsics.yml",
                bool no_display = false,
                string point_cloud_filename = "输出/point3D.txt"
                )
{
    //获取待处理的左、右相机图像
    int color_mode = 0;
    Mat rawImg = imread(fileList[picNum], color_mode);  //待处理图像 grayScale
    if(rawImg.empty()){
        std::cout <<"In Function stereoMatch, the Image is empty..."<< endl;
        return 0;
    }
    //截取
    Rect leftRect(0, 0, imgSize.width, imgSize.height);
    Rect rightRect(imgSize.width, 0, imgSize.width, imgSize.height);
    Mat img1 = rawImg(leftRect);                        //切分得到的左原始图像
    Mat img2 = rawImg(rightRect);                       //切分得到的右原始图像
    //图像根据比例缩放
if(imgScale != 1.f){
        Mat temp1, temp2;
        int method = imgScale < 1 ? INTER_AREA : INTER_CUBIC;
resize(img1, temp1, Size(), imgScale, imgScale, method);
        img1 = temp1;
resize(img2, temp2, Size(), imgScale, imgScale, method);
        img2 = temp2;
    }
    imwrite("输出/原始左图像.jpg", img1);
    imwrite("输出/原始右图像.jpg", img2);
```

```
    Size img_size = img1.size();

    //reading intrinsic parameters
    FileStorage fs(intrinsic_filename, CV_STORAGE_READ);
    if(!fs.isOpened())
    {
        std::cout <<"Failed to open file "<< intrinsic_filename << endl;
        return -1;
    }
    Mat M1, D1, M2, D2;                                    //左、右相机的内参数矩阵和畸变系数
    fs["M1"] >> M1;
    fs["D1"] >> D1;
    fs["M2"] >> M2;
    fs["D2"] >> D2;

    M1 *= imgScale;
    M2 *= imgScale;

    //读取双目相机的立体矫正参数
fs.open(extrinsic_filename, CV_STORAGE_READ);
    if(!fs.isOpened())
    {
std::cout <<"Failed to open file "<< extrinsic_filename << endl;
        return -1;
    }

    //立体矫正
    Rect roi1, roi2;
    Mat Q;
    Mat R, T, R1, P1, R2, P2;
    fs["R"] >> R;
    fs["T"] >> T;

    //Alpha 取值为 -1 时,OpenCV 自动进行缩放和平移
    cv::stereoRectify( M1, D1, M2, D2, img_size, R, T, R1, R2, P1, P2, Q, CALIB_ZERO_
DISPARITY, -1, img_size, &roi1, &roi2 );

    //获取两相机的矫正映射
    Mat map11, map12, map21, map22;
    initUndistortRectifyMap(M1, D1, R1, P1, img_size, CV_16SC2, map11, map12);
initUndistortRectifyMap(M2, D2, R2, P2, img_size, CV_16SC2, map21, map22);

    //矫正原始图像
    Mat img1r, img2r;
remap(img1, img1r, map11, map12, INTER_LINEAR);
remap(img2, img2r, map21, map22, INTER_LINEAR);
    img1 = img1r;
    img2 = img2r;

    //初始化 stereoBMstate 结构体
    StereoBM bm;

    int unitDisparity = 15;                    //40
    int numberOfDisparities = unitDisparity * 16;
```

```
    bm.state->roi1 = roi1;
     bm.state->roi2 = roi2;
     bm.state->preFilterCap = 13;
     bm.state->SADWindowSize = 19;                    //窗口大小
     bm.state->minDisparity = 0;                      //确定匹配搜索从哪里开始,默认值是 0
     bm.state->numberOfDisparities = numberOfDisparities;
                                                      //在该数值确定的视差范围内进行搜索
     bm.state->textureThreshold = 1000; //10          //保证有足够的纹理以克服噪声
     bm.state->uniquenessRatio = 1; //10              //!!使用匹配功能模式
     bm.state->speckleWindowSize = 200; //13          //检查视差连通区域变化度的窗口大小, 值
                                                        为 0 时取消 speckle 检查
     bm.state->speckleRange = 32; //32                //视差变化阈值,当窗口内视差变化大于阈
                                                        值时,该窗口内的视差清零,int 型
     bm.state->disp12MaxDiff = -1;

    //计算
    Mat disp, disp8;
     int64 t = getTickCount();
bm(img1, img2, disp);
     t = getTickCount() - t;
     printf("立体匹配耗时: %fms\n", t*1000/getTickFrequency());

    //将 16 位带符号的整型视差矩阵转换为 8 位无符号的整型矩阵
disp.convertTo(disp8, CV_8U, 255/(numberOfDisparities*16.));

    //视差图转为彩色图
    Mat vdispRGB = disp8;
    F_Gray2Color(disp8, vdispRGB);
    //将左侧矫正图像与视差图融合
    Mat merge_mat = F_mergeImg(img1, disp8);

    saveDisp("输出/视差数据.txt", disp);

    //显示
     if(!no_display){
        imshow("左侧矫正图像", img1);
        imwrite("输出/left_undistortRectify.jpg", img1);
        imshow("右侧矫正图像", img2);
        imwrite("输出/right_undistortRectify.jpg", img2);
        imshow("视差图", disp8);
        imwrite("输出/视差图.jpg", disp8);
        imshow("视差图_彩色.jpg", vdispRGB);
        imwrite("输出/视差图_彩色.jpg", vdispRGB);
        imshow("左矫正图像与视差图合并图像", merge_mat);
        imwrite("输出/左矫正图像与视差图合并图像.jpg", merge_mat);
cv::waitKey(10000);
        printf("\n 输入任意数字以继续\n");
        getchar();
         std::cout << endl;
     }
//cv::destroyAllWindows();

    //视差图转为深度图
    cout << endl <<"计算深度映射... "<< endl;
     Mat xyz;
     reprojectImageTo3D(disp, xyz, Q, true);          //获得深度图 disp: 720*1280
    cv::destroyAllWindows();
```

```
        cout << endl <<"保存点云坐标... "<< endl;
        saveXYZ(point_cloud_filename, xyz);

         cout << endl << endl <<"结束"<< endl <<"Press any key to end... ";

        getchar();
        return 0;
    }


    int main()
    {
        string intrinsic_filename = "intrinsics.yml";
         string extrinsic_filename = "extrinsics.yml";
        string point_cloud_filename = "输出/point3D.txt";

        /* 立体标定运行一次即可 */
    //initFileList("calib_pic", 1, 11);
    //stereoCalibrate(intrinsic_filename, extrinsic_filename);

        /* 立体匹配 */
        initFileList("test_pic", 1, 2);
        stereoMatch(0, intrinsic_filename, extrinsic_filename, false, point_cloud_filename);

        return 0;
    }
```

3. 运行结果

运行结果如图 3-76 所示，图中有相机标定参数，也有标定过程中出现的误差，当误差过大时，建议重新拍摄图像进行标定。图 3-77 为标定过程，图 3-78 为图像的视差图。通过视差图可以得到图像的深度，而最后图像的深度数据将会存储在如图 3-79 所示的.txt 文件中，前两列是点云的坐标，第三列表示深度，即物体每个点距摄像机的距离。



图 3-76 标定结果

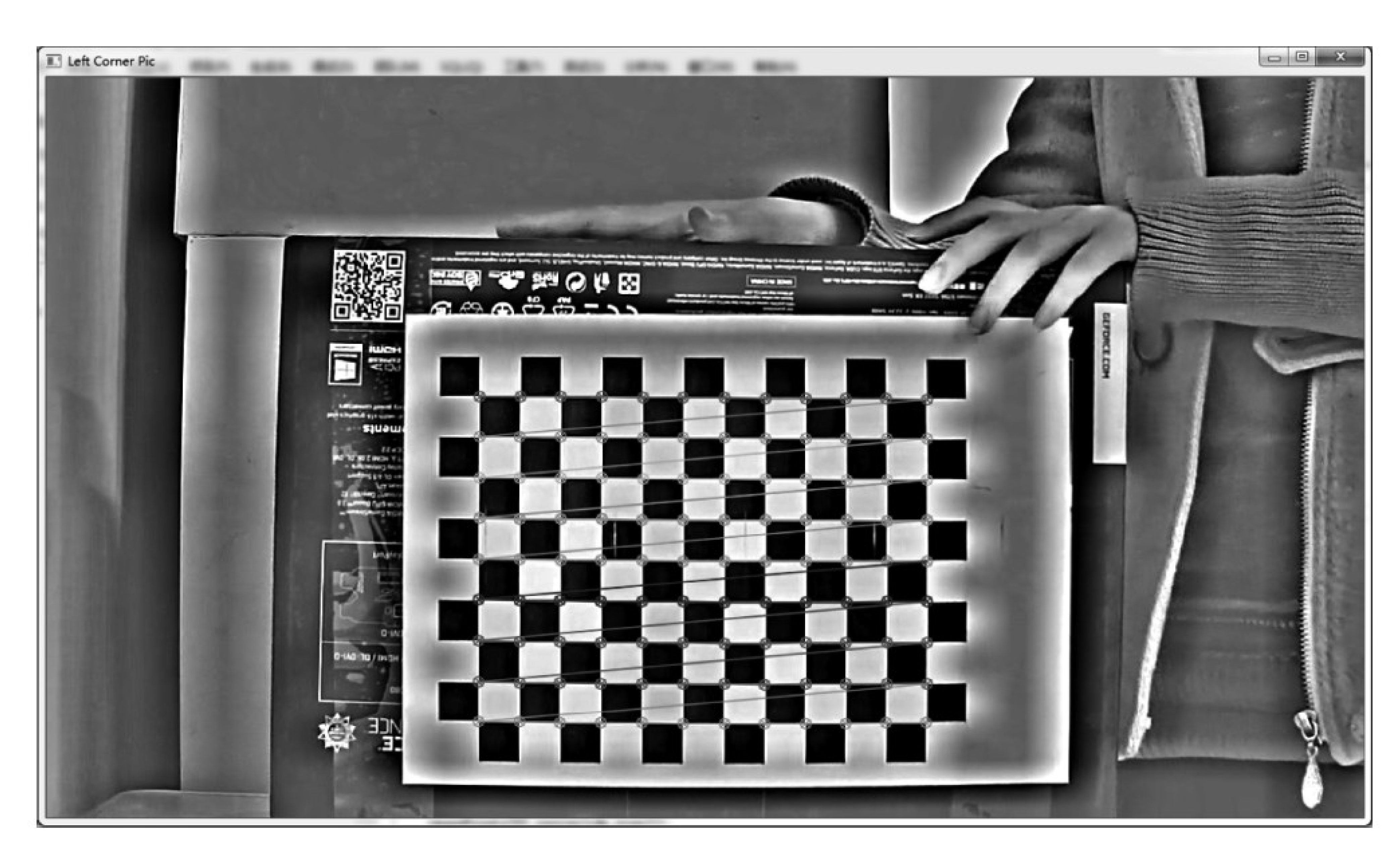


图 3-77　标定过程

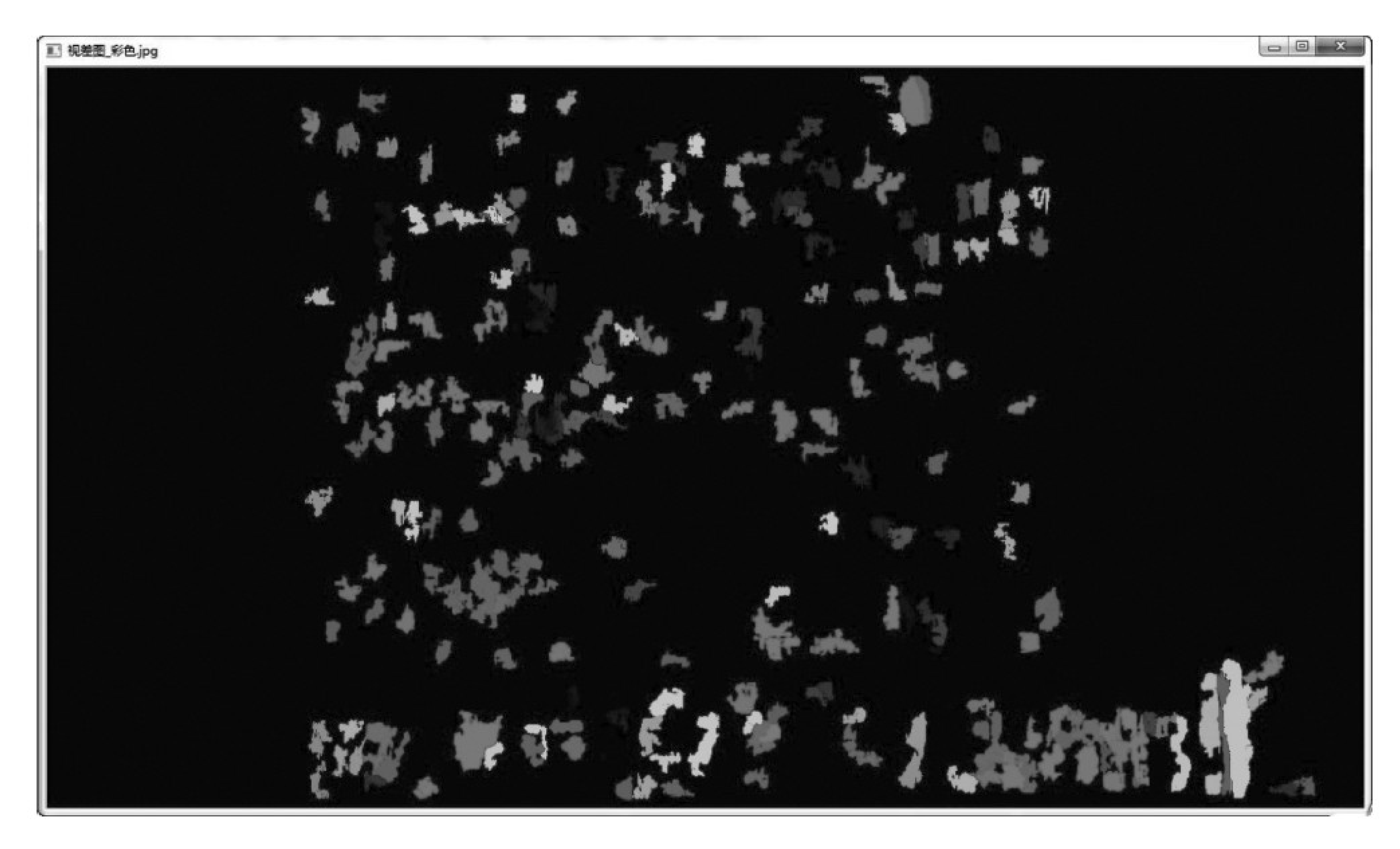
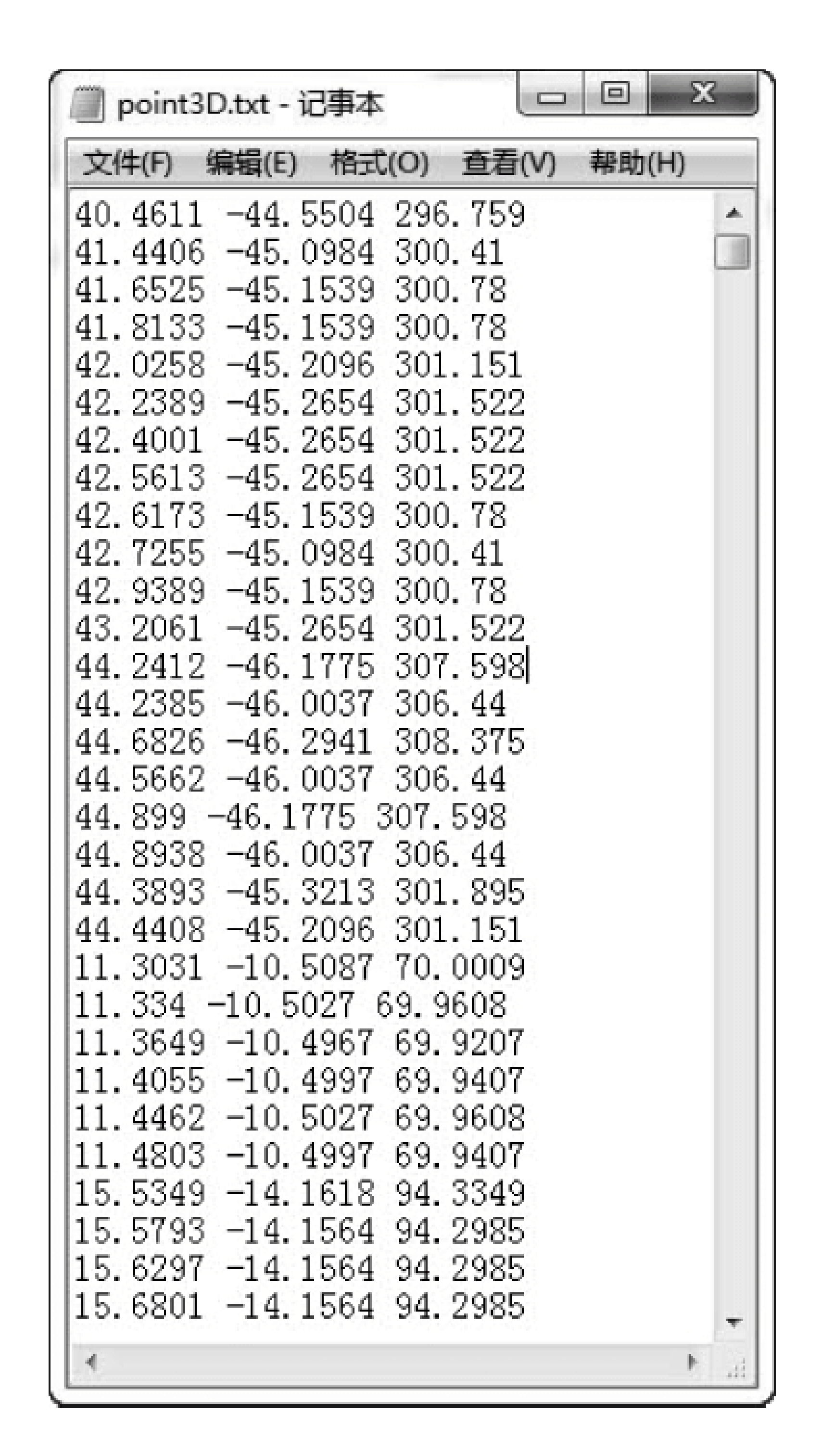


图 3-78　视差图




```
40.4611 -44.5504 296.759
41.4406 -45.0984 300.41
41.6525 -45.1539 300.78
41.8133 -45.1539 300.78
42.0258 -45.2096 301.151
42.2389 -45.2654 301.522
42.4001 -45.2654 301.522
42.5613 -45.2654 301.522
42.6173 -45.1539 300.78
42.7255 -45.0984 300.41
42.9389 -45.1539 300.78
43.2061 -45.2654 301.522
44.2412 -46.1775 307.598
44.2385 -46.0037 306.44
44.6826 -46.2941 308.375
44.5662 -46.0037 306.44
44.899 -46.1775 307.598
44.8938 -46.0037 306.44
44.3893 -45.3213 301.895
44.4408 -45.2096 301.151
11.3031 -10.5087 70.0009
11.334 -10.5027 69.9608
11.3649 -10.4967 69.9207
11.4055 -10.4997 69.9407
11.4462 -10.5027 69.9608
11.4803 -10.4997 69.9407
15.5349 -14.1618 94.3349
15.5793 -14.1564 94.2985
15.6297 -14.1564 94.2985
15.6801 -14.1564 94.2985
```

图 3-79 深度数据

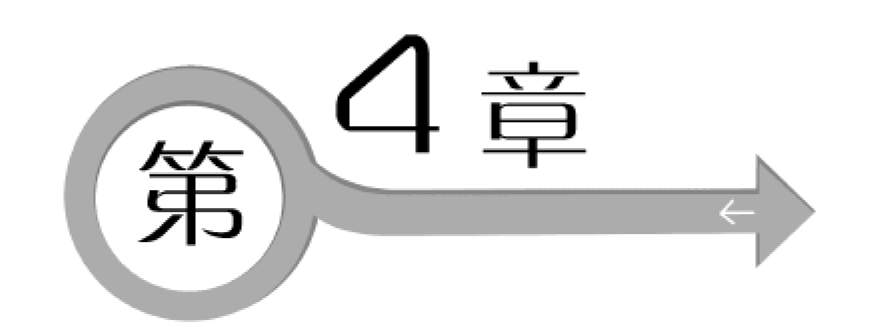
3.6 本章小结

本章主要介绍了几个常用的图像处理算法并通过 OpenCV 进行了实现。3.1 节主要介绍了 OpenCV 最常用的 Mat 类和基础的输入输出函数。3.2 节介绍了反向算法,并通过反向算法对 OpenCV 常用的函数和处理方式进行了解析,为后续的学习扫清障碍。3.3 节主要介绍了 OpenCV 处理图像时常用的 addweight()函数。3.4 节介绍了三种常见的图像去噪算法,其中均值滤波和高斯滤波采用了 OpenCV 自带的函数。3.5 节主要介绍了机器视觉中比较热门的双目视觉技术,首先介绍了双目视觉的基本原理,之后介绍了如何使用 OpenCV 实现双目摄像机的标定,最后使用 OpenCV 实现了双目视觉测试物体的深度。

3.7 参考文献

[1] http://lib.csdn.net/article/opencv/24583.
[2] http://blog.csdn.net/giantchen547792075/article/details/7061391/.
[3] http://blog.sina.com.cn/s/blog_6a0e04380100r289.html.
[4] http://blog.csdn.net/qianqing13579/article/details/45318279.
[5] http://blog.csdn.net/poem_qianmo/article/details/20537737.
[6] http://www.bubuko.com/infodetail-1179988.html.
[7] http://baike.baidu.com/view/1485502.htm.

[8] 王智文，李绍滋. 基于多元统计模型的分形小波自适应图像去噪[J]. 计算机学报，2014，37(6)：1380-1389.
[9] 王科俊，熊新炎，任桢. 高效均值滤波算法[J]. 计算机应用研究，2010，27(2)：434-438.
[10] 许景波. 高斯滤波器逼近理论与应用研究[D]. 哈尔滨：哈尔滨工业大学，2007.
[11] 王振雷. 基于图像N-邻域的协同滤波去噪方法的研究[D]. 西安：西安电子科技大学，2014.
[12] 郭红涛，王小伟，章勇勤. 广义非局部均值算法的图像去噪[J]. 计算机应用研究，2015，32(7)：2218-2221.
[13] 王新然. 基于计算机视觉的物体体积测量系统[D]. 大连：大连理工大学，2012.
[14] Harwerth R S，Boltz R L. Behavioral measures of stereopsis in monkeys using random dot stereograms[J]. Physiology & Behavior，1979，22(2)：229-234.



GPU和CUDA的介绍和应用

相信读者通过第 1 章的阅读可以对 GPU 和 CUDA 有了大体了解,本章将详细介绍 GPU 的架构和内部结构、如何搭建 CUDA 环境、CUDA C 语言的最小扩展集、CUDA 的线程层次和 GPU 的寄存器等。

4.1　CUDA 的介绍

统一计算设备架构(Compute Unified Device Architecture,CUDA)是 NVIDIA(英伟达)公司在 2007 年 6 月提出的一种全新的并行计算架构,后发展成一款独立的软件,它可以基于硬件设备 GPU 实现数据的并行化处理。CUDA 是使原本只能处理图像显示数据的 GPU 可以通过 CUDA 处理各种数据,分担 CPU 的工作量,实现 GPU-CPU 异构共同处理数据,最终实现计算机硬件统一化处理数据的体系。如图 4-1 所示为 CUDA 的商标。

CUDA 相当于开发平台上的一个函数库,在平台上安装这个库即可让 GPU 工作。它好比在计算机上安装一个 Visual Studio 编译平台,调用自带的库即可编写 C++(如图 4-2 所示),或者在计算机上安装 Eclipse 编译平台,调用自身的库即可编写 Java(如图 4-3 所示)。同理,在使用 GPU 之前需要在计算机上先安装 C++编译平台 Visual Studio,然后在这个平台上安装 CUDA,这样就可以让 GPU 从原本图像处理工作中解放出来,与 CPU 实现异构,对数据进行处理,二者共同工作,如图 4-4 所示。



图 4-1　CUDA 商标

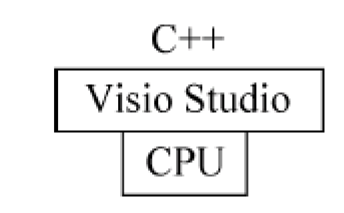


图 4-2　VS 编写 C++

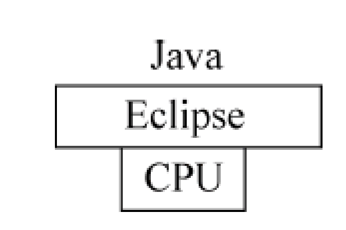


图 4-3　Eclipse 编写 Java

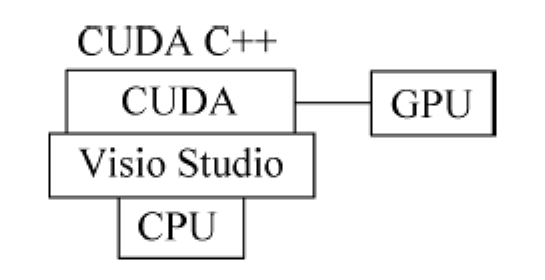


图 4-4　VS 和 CUDA 驱动 GPU

CUDA 一经推出便受到极大的欢迎，并且迅速在各个领域中取得了应用。

在消费级市场上，几乎每一款重要的消费级视频应用程序都已经使用 CUDA 加速或很快将会利用 CUDA 来加速，其中不乏 Elemental Technologies 公司、MotionDSP 公司以及 LoiLo 公司的产品。大家耳熟能详的阿里云技术，也采用了 CUDA 的 GPU 加速。在互联网市场上，CUDA 已经被大量使用。

在科研界，CUDA 一直受到热捧。例如，CUDA 现已能够对 AMBER 进行加速。AMBER 是一款分子动力学模拟程序，全世界在学术界与制药企业中有超过 60 000 名研究人员使用该程序来加速新药的探索工作。

在金融市场，Numerix 以及 CompatibL 针对一款全新的对手风险应用程序发布了“CUDA 支持”并取得了 18 倍速度提升。Numerix 被近 400 家金融机构广泛使用。

CUDA 的广泛应用造就了 GPU 计算专用 Tesla GPU 的崛起。全球财富五百强企业现在已经安装了 700 多个 GPU 集群，这些企业涉及各个领域，例如能源领域的斯伦贝谢与雪佛龙，以及银行业的法国巴黎银行。

支持 CUDA 的硬件设备有 NVIDIA 的 GeForce 系列、ION、Tesla 等，并且 CUDA 现已能在 Windows、Linux 和 Mac 三种系统上完美运行。目前，CUDA 系列的最高版本是 CUDA 8.0。

本书在使用 CUDA 时选用的是 CUDA 6.5，不过下载时需要注意 CUDA 的版本，在 CUDA 6.5 及以前的版本会有笔记本(notebook)版本和台式机两种版本，在 CUDA 7.0 以后则没有这种分类了，所以下载时一定注意版本型号。

4.2　GPU 的内部结构

了解 GPU 内部结构对提高 GPU 的利用率非常重要，并且对后续学习如何开发 GPU 会有巨大的帮助。

4.2.1　GPU 内部结构的简单介绍

GPU 内部结构原理图比较抽象，为帮助没有任何 GPU 基础的读者更好地理解该内容，本节先做一些简单说明。已有相关基础知识的读者，可以直接跳过本节，进入 4.2.2 节进行具体内部结构的学习。

GPU 是由无数个简单的计算节点高度集成的一个芯片，这些节点之间互不影响，独立地进行浮点计算，通常称这些节点为线程(thread)，如图 4-5 所示。

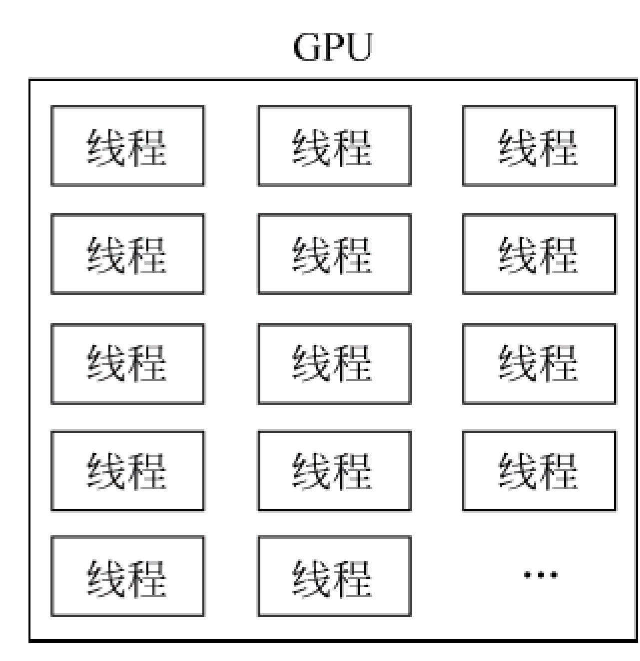


图 4-5　GPU 内部结构(简单版)

每个线程之间相互不影响，所有线程集成在一起就形成了一块完整的GPU。其中，一定数量的线程集合在一起成为一个“线程块”（block），一定数量的线程块聚集在一起成为“线程网格”，通常情况下会将其简称为“线程格”（grid），如图4-6所示为GPU内部结构简图，实际上每个线程块包含的线程远远不止这些，线程格也不是只包含这两个线程块。

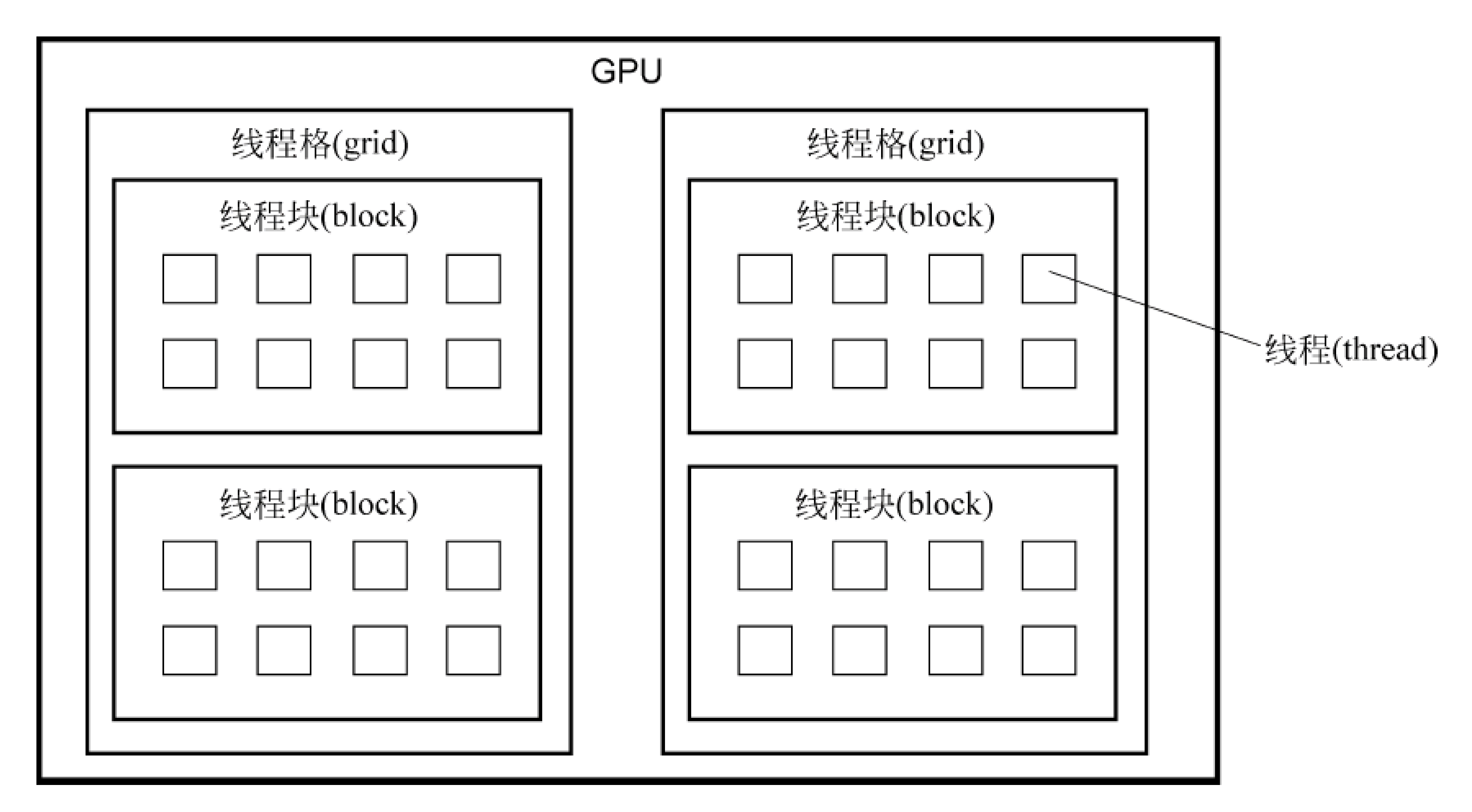


图4-6　GPU内部结构图

这些线程是如何使用的？下面通过一个简单的例子进行说明。现在有两个数组，每个数组中都有八个数字，还有一个空数组如下：

```
a = [0,1,2,3,4,5,6,7];
b = [10,11,12,13,14,15,16,17];
c = [];
```

要求将每组数据的对应位置的两个数字相加并存在数组c的对应位置中。因为每一个线程都有单独的计算能力，所以让每一个线程都计算其中一部分，即每一个线程单独完成一次加法运算，那么现在准备启动八个线程，完成这次计算。

可以在一个线程格（grid）中取其中一个线程块（block），并在该线程格中启动八个线程，每个线程进行一次加法计算，相互之间没有影响，进行的模式如图4-7所示。

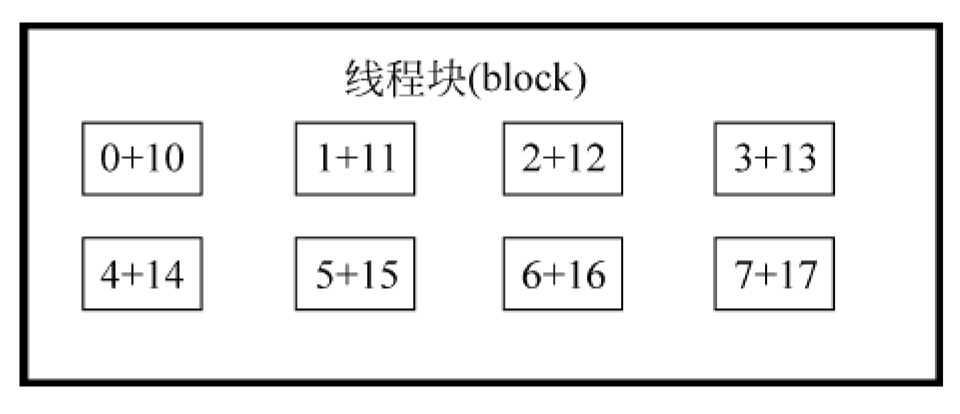


图4-7　计算示意图1

当然，如果愿意，也可以使用同一个线程格中的两个线程块里的线程来做这次计算，计算效果如图4-8所示。

同样，在线程够用情况下，甚至可以启用多个线程格，每个线程格中启动一定数量的线程，来完成这次计算，但是要求每个线程块中启动的线程数量是相同的。比如选用两个线程格，每个线程格开启两个线程块，每个线程块中开启两个线程来计算，如图4-9所示。

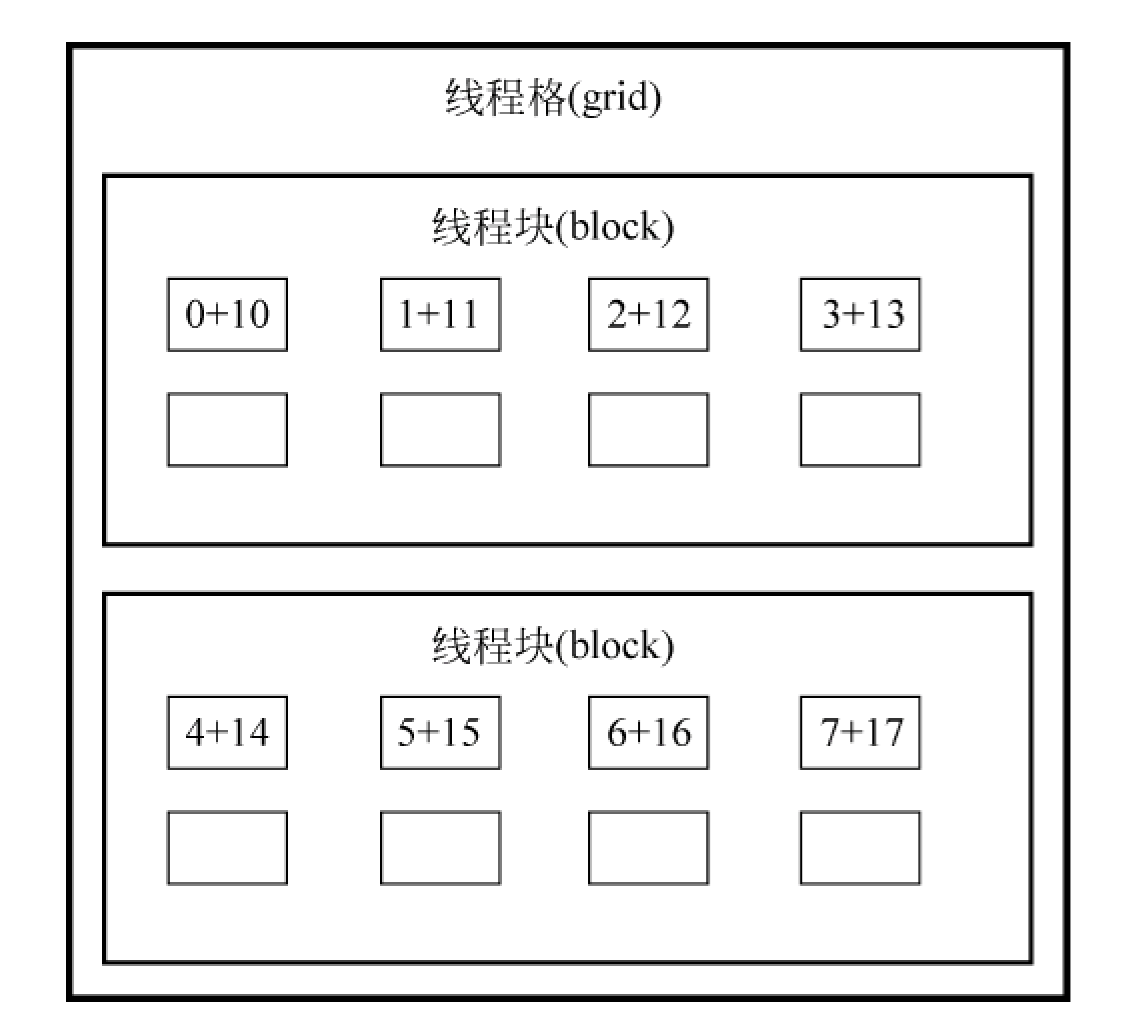


图 4-8 计算示意图 2

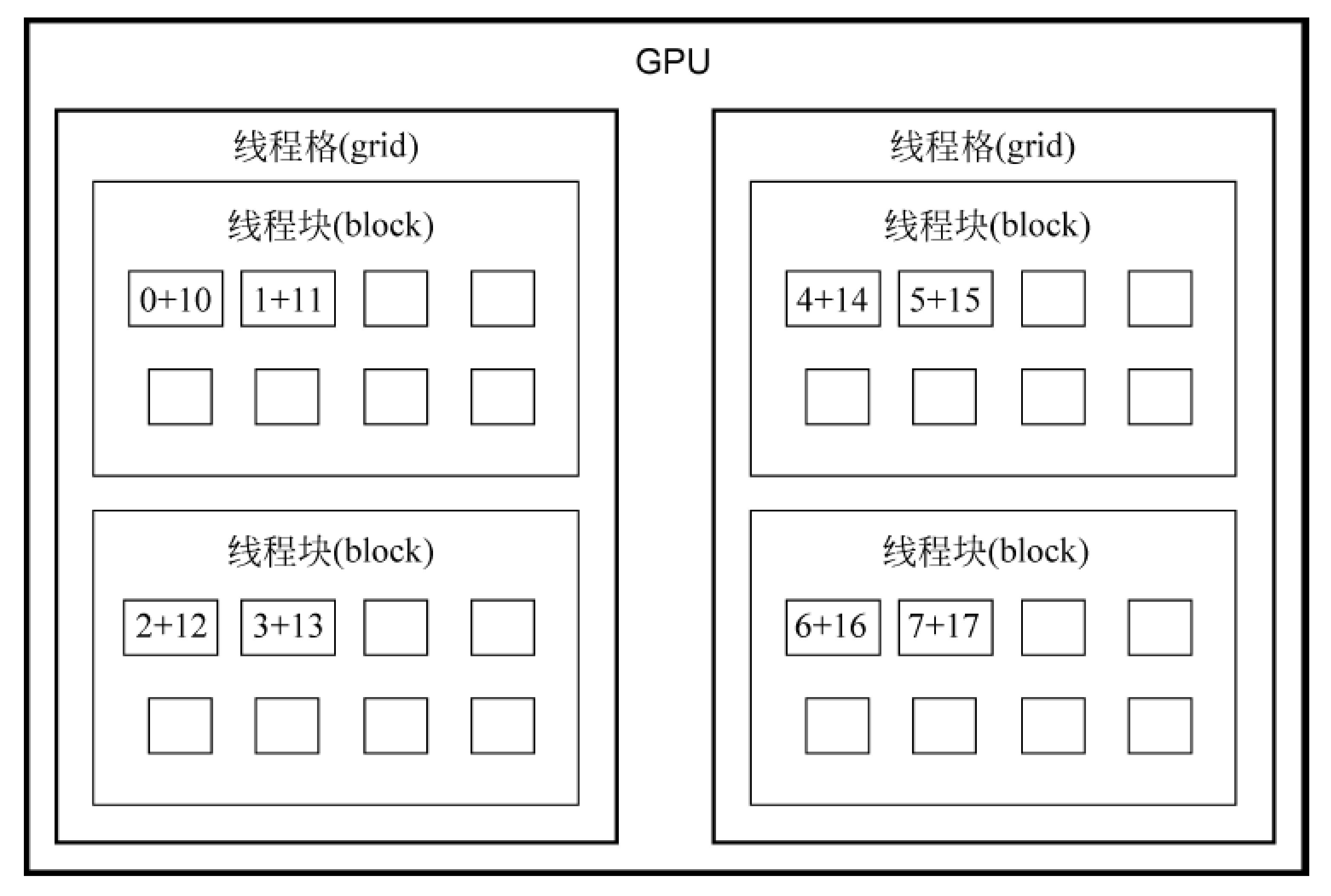


图 4-9 计算示意图 3

在分配线程时，每一次只能确定开启多少个线程，比如在前面的例子中，如果选择开启四个线程块，那么仅能计算得到前四项。如果开启了很多线程块，不但会造成资源浪费，还会将原本存在寄存器中的数据放在缓存中，降低了使用效率，增加了处理时间。所以通常情况下，会根据实际情况合理地分配线程实现最大利用率。

以上介绍的都是一些简单的内部结构和分配线程方式，仅能作为 GPU 的基础入门知识，却不足以支撑 GPU 的开发，因为有效的 GPU 开发需要更系统的、更细节的 GPU 内部结构信息，这些内容将在后续章节中介绍。不过相信 GPU 初学者经过本节的学习会更容易理解 4.2.2 节的内容。

4.2.2 GPU 的架构

本节将详细介绍 NVIDIA 公司的 GPU 架构。在 NVIDIA 公司的 GPU 发展过程中，

多种不同架构被先后采用，本节将讲解其中几种比较典型的架构[1]。

1. G80 架构

G80 架构是 NVIDIA 公司在 2006 年 11 月推出的第一批统一架构 GPU，其整合了 GeForce 8800 GTX 核心，768MB GDDR3 高速内存以及 128 个流处理器（Stream Processor，SP），其结构如图 4-10 所示。

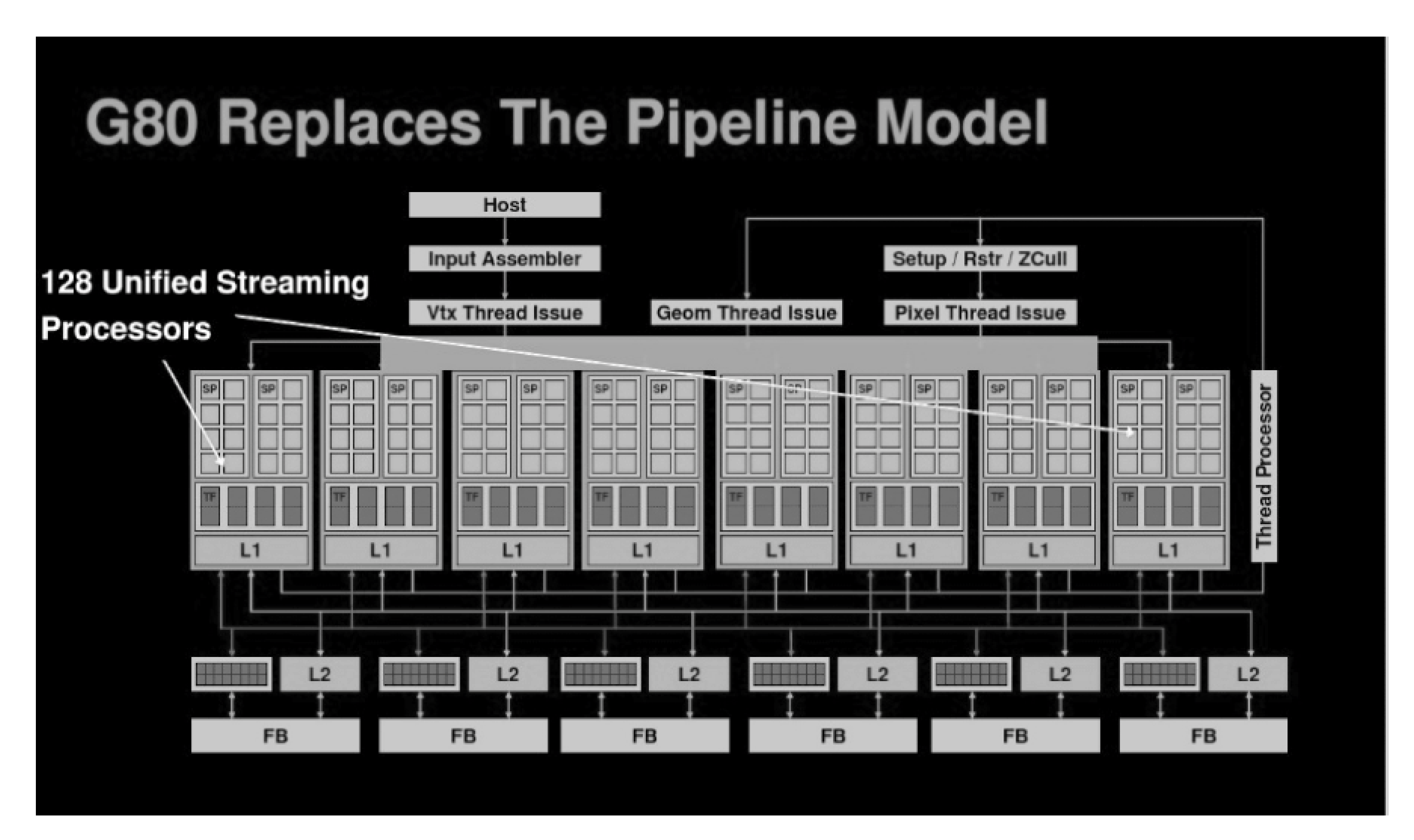


图 4-10 G80 架构

统一结构中的基本单元被称为流多处理器（Streaming Multiprocessor，SM）。流多处理器是 GPU 中最底层的独立硬件结构，可以将其看成一个 SIMD 处理单元。G80 架构一共有 16 个流多处理器，每个流多处理器又包括八个流处理器（Streaming Processor，SP）和两个特殊处理单元（Special Function Unit，SFU）。此外，每个流多处理器都包含一个 16KB 大小的共享存储器（Shared Memory）和 8192 个 32 位寄存器，其详细结构如图 4-11 所示。

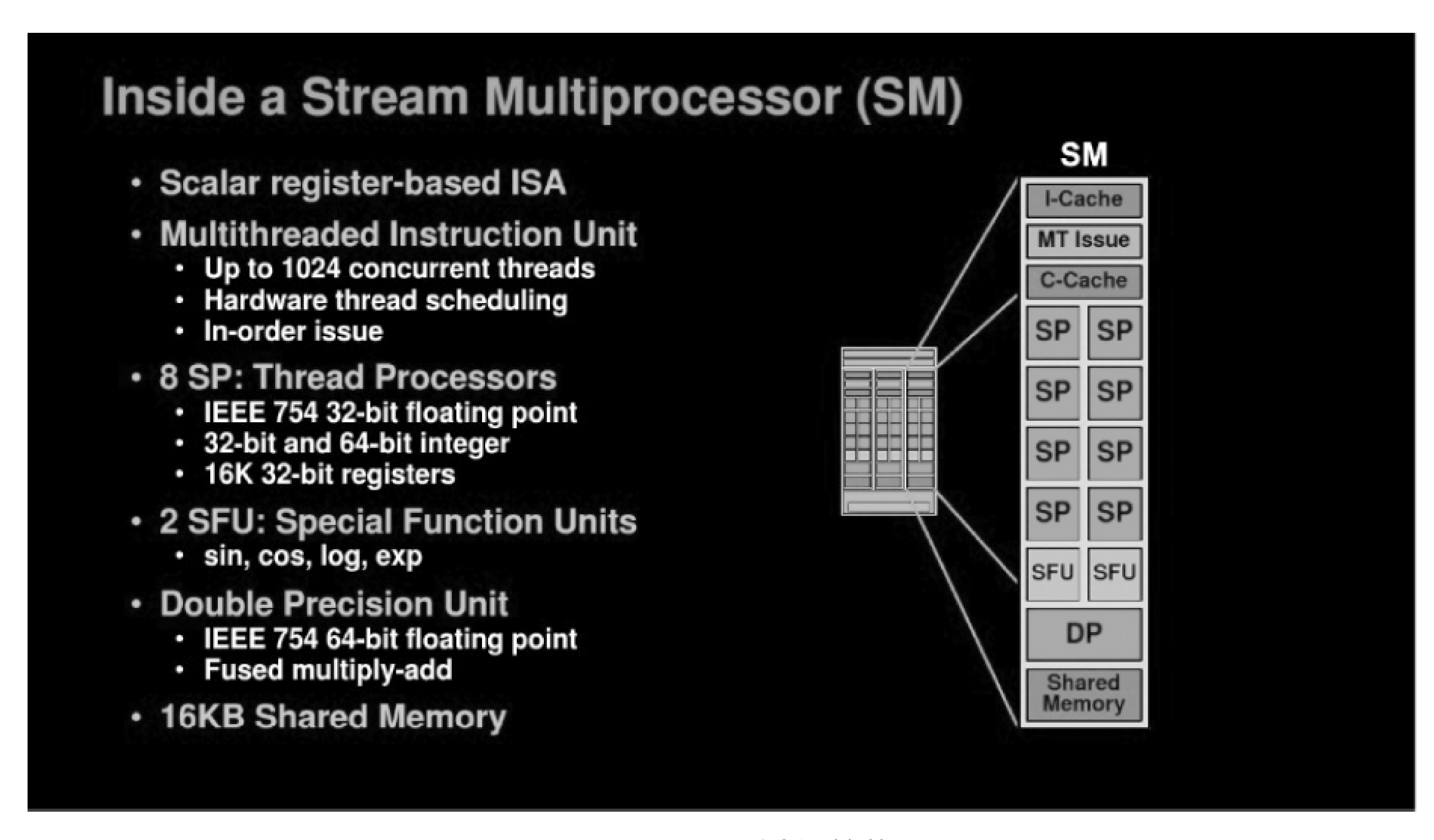


图 4-11 G80 详细结构

GPU 中每个流多处理器都支持成千上万的线程并行执行，每次 kernel 函数启动之后，线程(thread)会被分配到对应的流多处理器中执行。虽然大量的线程会被分配到不同的流多处理器，但是同一个线程块(block)中的线程一定会在同一个流多处理器中被并行执行。

而使用开发 GPU 的 CUDA 则采用了 SIMT(Single Instruction Multiple Thread)架构来管理和执行这些线程。这些线程以 32 个为一组，构成了一个单元，这些单元称为 warp。warp 中所有的线程并行执行同一个指令，每个线程拥有它自己的状态寄存器，并且用该线程内部的独有数据执行指令。

但是这种执行机制很容易造成执行的速度不一致，其原因是大部分的线程只是逻辑上的并行，并不是所有的线程都可以在物理上同时执行。因此，如何合理地同步所有的线程也是开发中需要注意的一个重点。

2. GT200 架构

GT200 架构是 NVIDIA 公司推出的第二批统一架构 GPU，该架构在 G80 的基础上做了拓展，主体结构很相似，但其功能要比 G80 架构增强了一些，可以理解为 G80 的升级版，其升级的主要部分为：

- 流多处理器的数目由 128 个增加到 240 个；
- 每个 TPC 所包含的流多处理器数量由两个增加到三个；
- 每个流多处理器中所包含的寄存器数目由 8192 个增加到 16 384 个；
- 每个流多处理器中添加了一个双精度浮点运算单元；
- 前端总线位宽由 328 位增加到 512 位；
- 带宽由 86GB/s 增加到 142GB/s；
- 与主机端的数据传输接口由 PCIe 1.0 x 16 提升为 PCIe 2.0 x 16。

GT200 的架构图如图 4-12 所示。

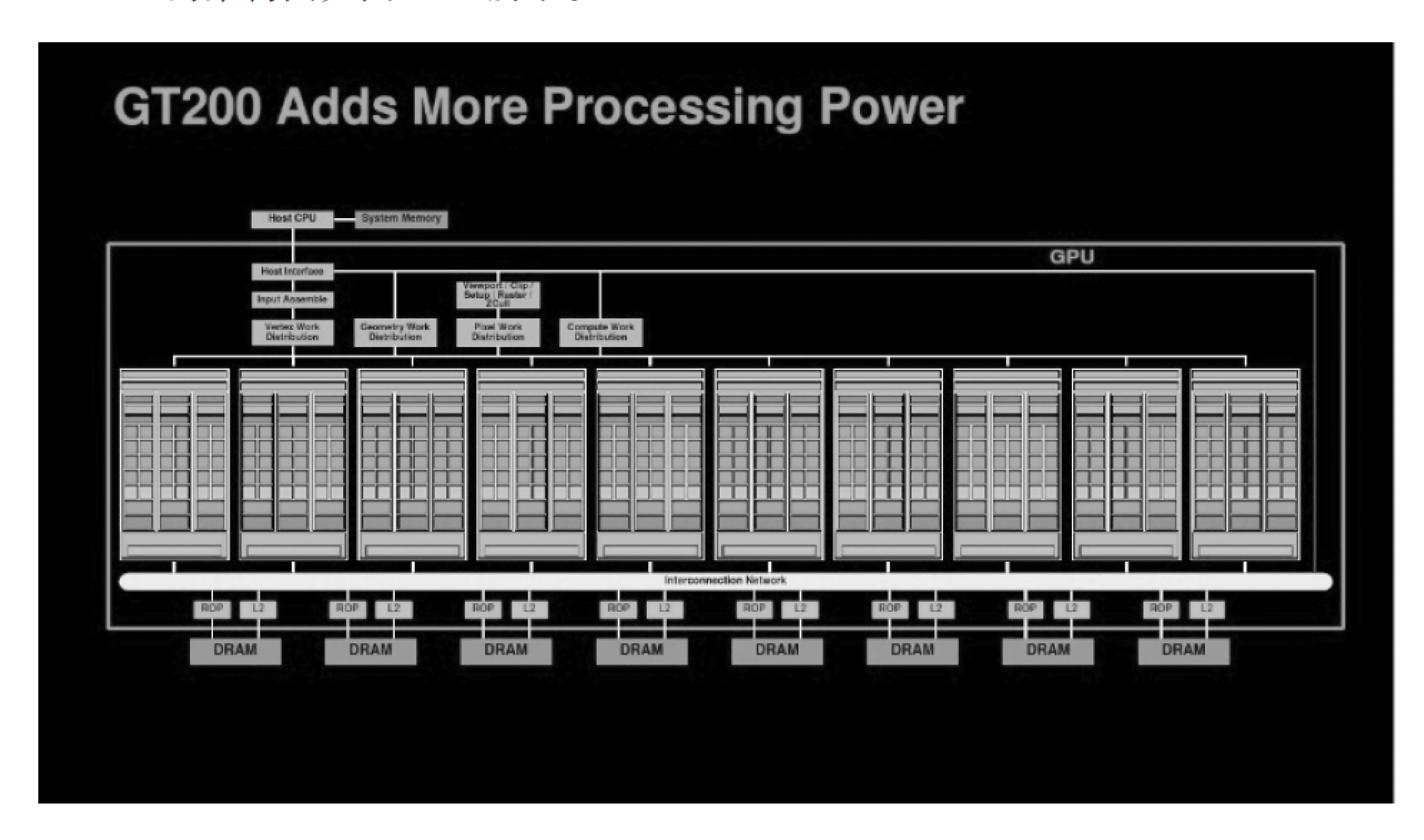


图 4-12　GT200 架构

3. Fermi 架构

Fermi 架构是 NVIDIA 公司推出的第三批统一架构 GPU，可以说是第一个完整的 GPU 计算架构。相比于 G80 和 GT200，该架构发生了较大的改变，其浮点计算效率大幅提升，使整个 GPU 得到了质的飞越。其架构如图 4-13 所示。

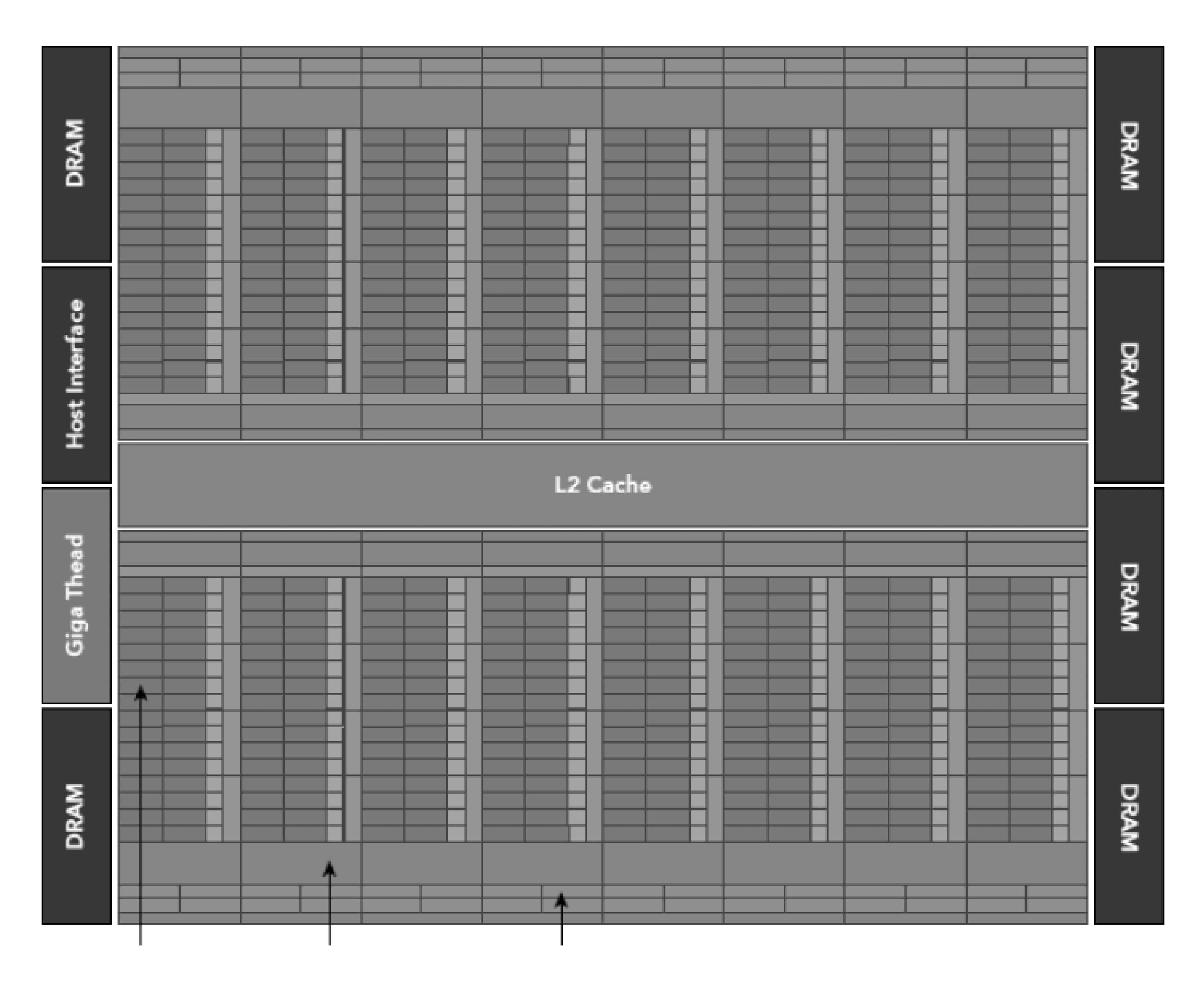


图 4-13　Fermi 架构

Fermi 架构的工作模式和主要提升如下：

- 512 个加速器核心(accelerator core)即 CUDA core；
- 16 个流多处理器，每个流多处理器包含了 32 个 CUDA core；
- 6 个 384 位的 GDDR5 DRAM；
- GigaThread engine 将线程和线程块分配给流多处理器调度；
- 每个流多处理器有 16 个读取/存储(load/store)单元，允许每个时钟脉冲(clock cycle)为 16 个线程计算原地址和目的地址；
- 每个流多处理器包含两个 Warp Scheduler 和两个指令发送单元(Instruction Dispatch Unit)，每个线程块被分配到一个流多处理器中，所有该线程块中的线程被分配到不同的 warp 中；
- 每个流多处理器同时可处理 48 个 warp。

4. Kepler 架构

Kepler 架构是 NVIDIA 公司推出的第四批统一架构 GPU，相比于 Fermi，其架构效率更高，性能更好，架构图如图 4-14 所示。

Kepler 架构的主要提升如下：

- 提升到 15 个流多处理器；
- 有 6 个 64 位内存控制器；
- 有 192 个单精度 CUDA corse、64 个双精度单元、32 个特殊处理单元、32 个读取/存储单元；

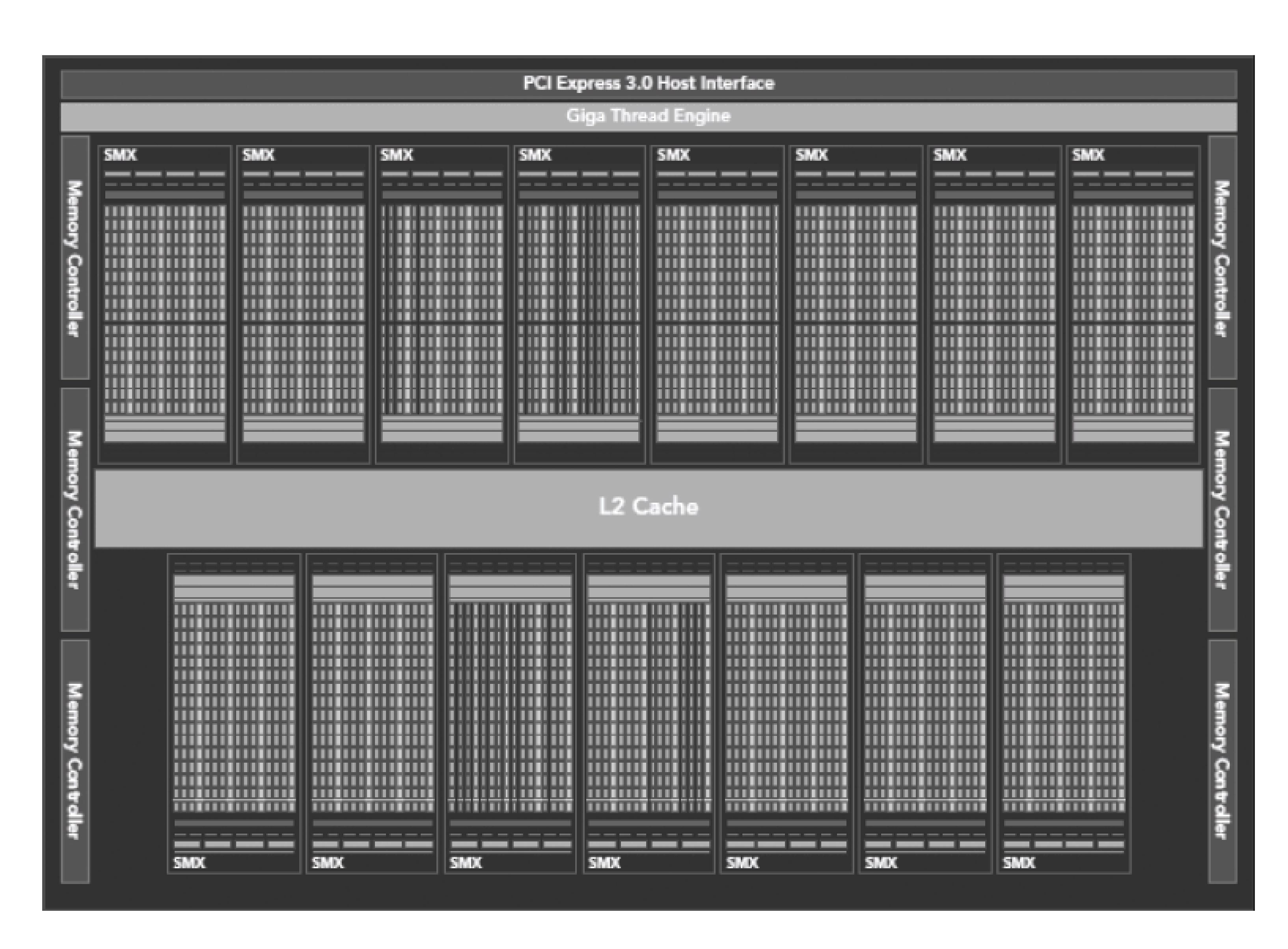


图 4-14　Kepler 架构

- 寄存器组(Register File)增加到 64KB;
- 每个 SM 最多可以同时调度 64 个 warp,共计 2048 个线程。

随着 GPU 的日益发展,更新的架构被不断地设计出来,NVIDIA 公司的老对手 AMD 公司也推出了很多优秀的架构,并且每一个新架构都会使 CPU 的功能提高很多。这促使 GPU 开发者不断更新对 GPU 架构内部结构和工作原理的了解,以便在开发中更好地优化程序。

4.2.3　常见 GPU 的挑选

GPU 越来越火,带来的不只是 GPU 的更新换代,还有对 GPU 性能五花八门的解释,越来越多的说法让人眼花缭乱,对于一个新人来说如何选择合适的 GPU 也是一个比较难的问题。本节将针对 NVIDIA 公司近几年推出的 GPU 进行简单的性能介绍,为读者挑选合适的 GPU 提供参考。

有很多因素会影响 GPU 的运行速度,比如 CUDA 核、时钟的速度、RAM 的大小,但是这些带来的影响比较微弱,而真正最影响速度的则是显存的带宽,所以选择 GPU 时可以用带宽来衡量一下 GPU 的速度。图 4-15 为 NVIDIA 系列 GPU 和常见 CPU 的带宽比较。

在芯片架构相同时,带宽可以直接进行对比。比如,Pascal 显卡 GTX 1080 和 1070 的性能对比,只需看显存带宽。GTX 1080(320GB/s)比 GTX 1070(256GB/s)快 25%。但是芯片架构不同,则不能直接对比,比如 Pascal 和 Maxwell(GTX 1080 和 Titan X),不同生产工艺下带宽相同,但不能保证二者运行的效果完全一样。

不同应用领域考虑 GPU 的种类也不相同,比如近几年很火热的深度学习,其关注点在于 GPU 是否兼容 cuDNN。绝大多数深度学习要用 cuDNN 来做卷积,因此 Kepler 或更好的 GPU,即 GTX 600 系列或以上效果会更好。如果仅仅通过深度学习训练速度来判断 GPU 的强弱,则可以参考图 4-16。



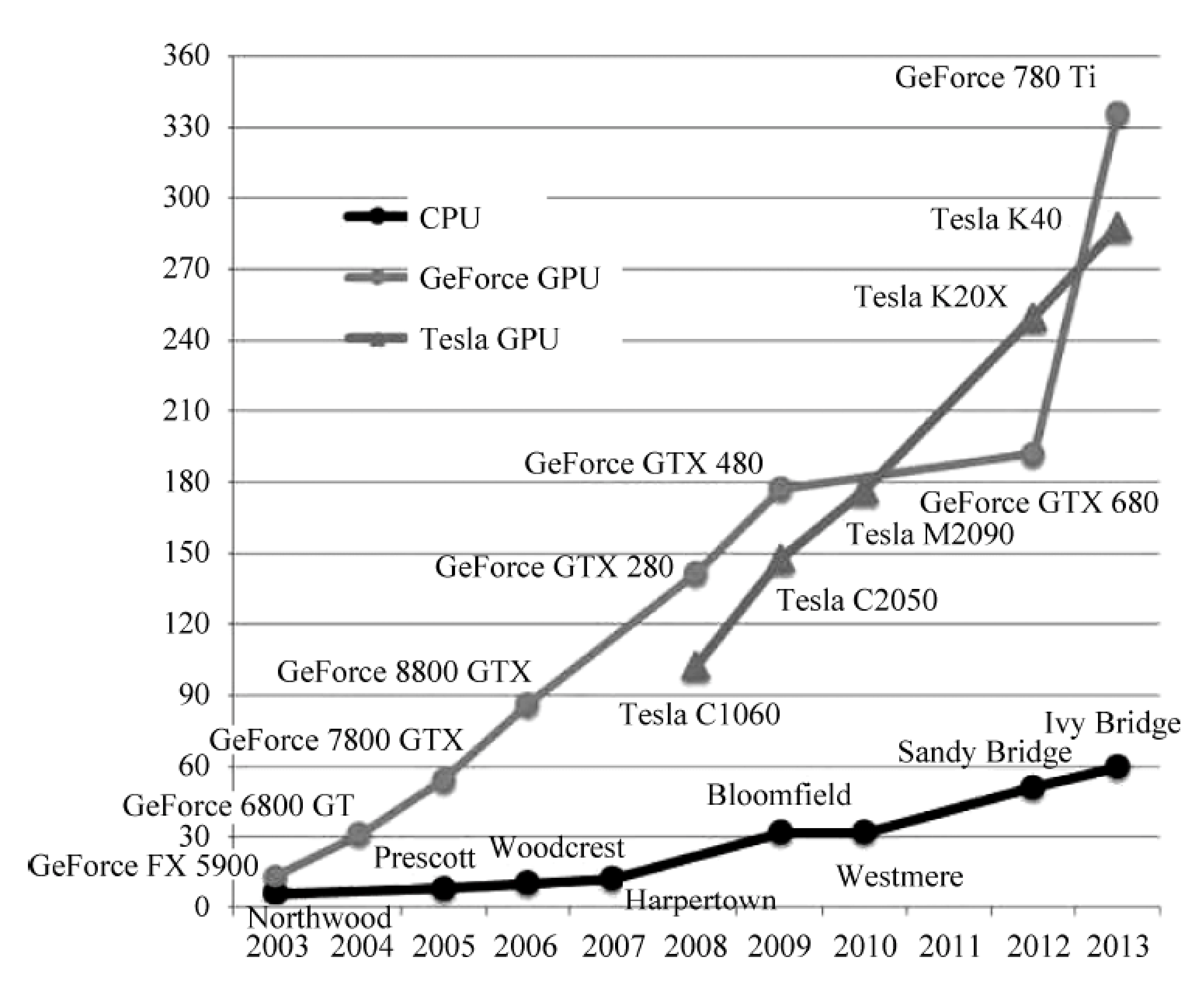


图 4-15 GPU 与 CPU 带宽比较

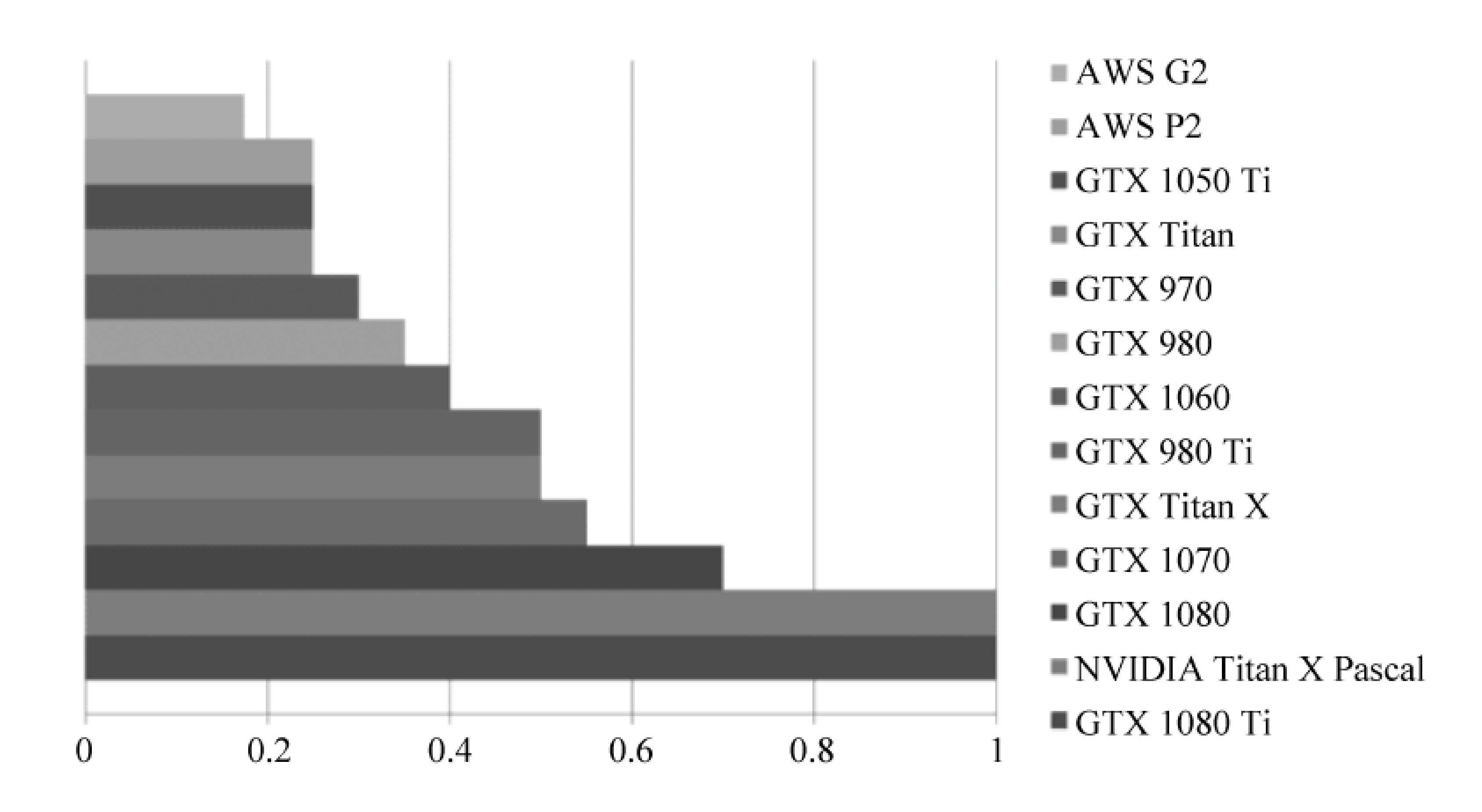


图 4-16 深度学习 GPU 处理速度

在做并行图像处理时，也可以参考深度学习的处理速度来合理地挑选 GPU 的种类和型号。总结如下：

- 并行处理能力强：Tesla K40/K80；
- 综合能力强：Titan X Pascal、GTX 1080Ti；
- 性价比高：GTX 1060；
- 普遍使用：GTX 600 及以上。

4.3　并行处理介绍

并行处理，顾名思义，在同一时刻同时处理多件事情，比如在语音通话时记录数据同时处理相关文件。并行处理的主要目的是节省解决复杂问题的时间。对于计算机，并行处理同样是指计算机系统能同时执行两个或更多个任务。

在计算机中，传统的处理方式是使用 CPU 来做所有的工作，但 CPU 是传统的串行处理，虽然 CPU 的处理能力已经发展得非常强大，但是仍然不能摆脱串行处理的瓶颈：“当计算量足够大时，仍然需要大量的时间。”就好比是一个生产线上的熟练工人，即便处理速度非常快，也无法独自完成生产线上成千上万的产品。

这时人们突然发现了每个计算机中都有的 GPU。随着一代又一代的升级，CPU 已经有了非常强大的浮点型数据处理能力。但是不是所有的数据都可以使用 GPU 进行处理，它只能处理图像数据，于是开发人员就想将 CPU 需要处理的数据伪装成图像信息并在 GPU 中运行，最后得到了意想不到的结果，处理速度得到了巨大的提升。

如果说 CPU 是一个非常熟练的工人，那么 GPU 中间就集成了成千上万的初级工人，每个人仅仅负责一个产品，只要给所有的初级工人做一个示范，所有的工人，就都可以模仿示范，做好手中的那件商品，最后再把所有的商品交给公司即可完成任务，如图 4-17 和图 4-18 所示。





图 4-17　CPU

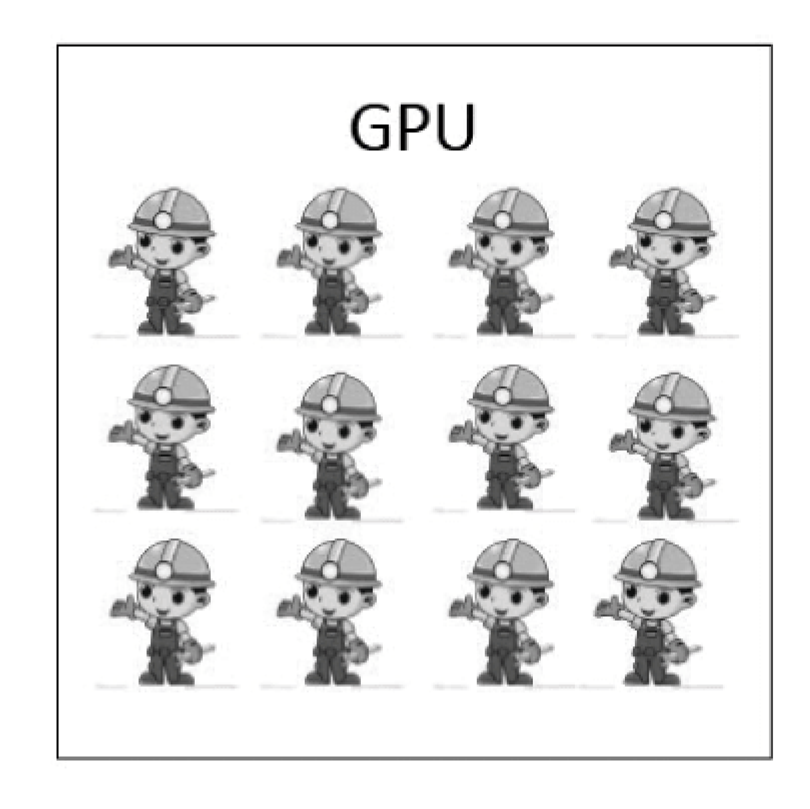


图 4-18　GPU

使用程序打一个比方，现在想把两个一维向量 a 和 b 中的数据进行求和，并在屏幕中显示出来，那么 CPU 中的处理方式就是将两个向量中的数据依次进行求和并显示出来，如图 4-19 所示。

```
a=[1,2,3,4,5];
b=[6,7,8,9,10];
```

而对于 GPU，这个计算过程是：CPU（熟练工人）给 GPU（一群初级工人）分配好任务，然后 GPU 共同执行所有任务，高效快捷地完成所有任务后，把结果返还给 CPU，完成任务，如图 4-20 所示。

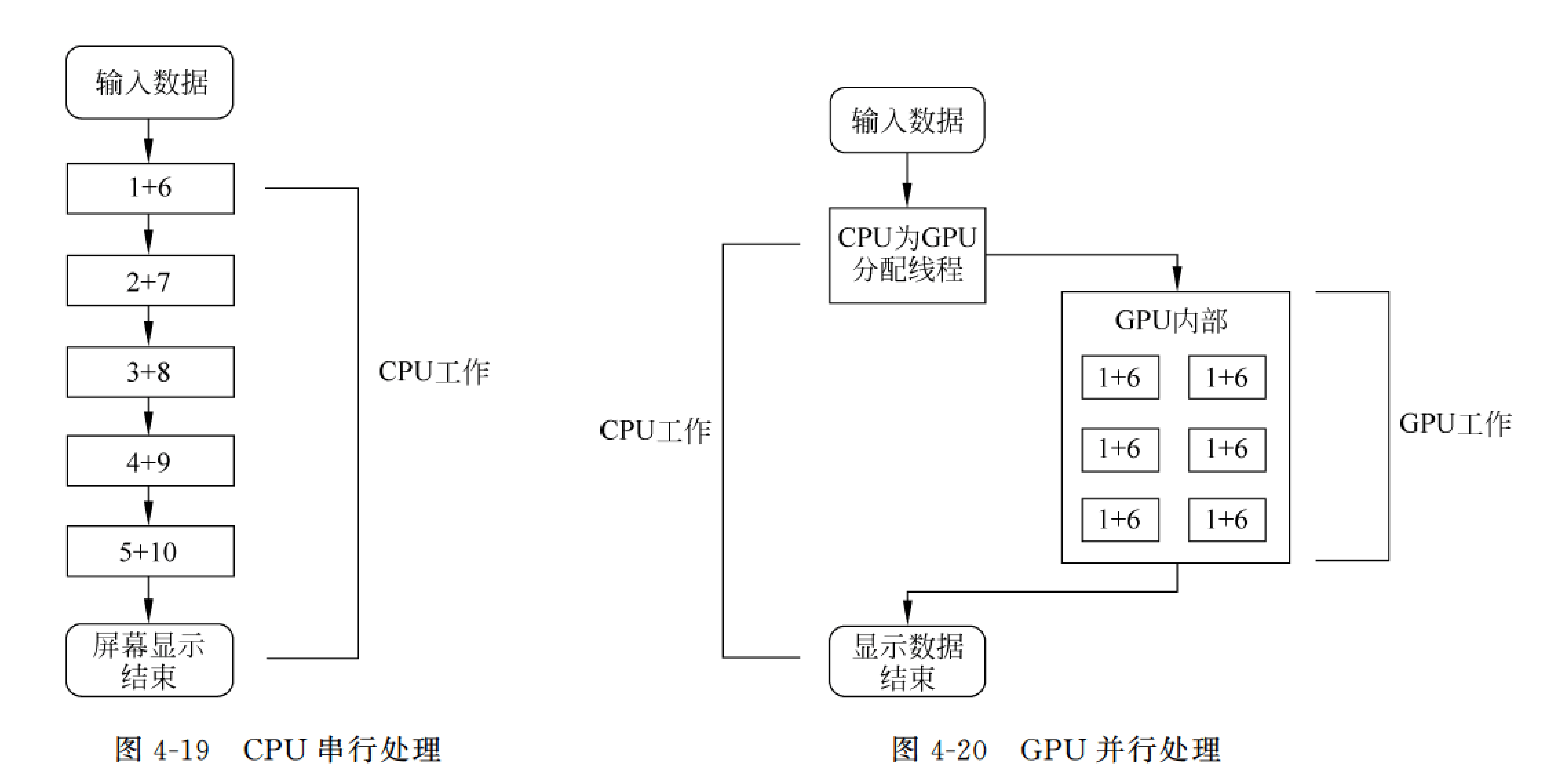


图 4-19　CPU 串行处理　　　　图 4-20　GPU 并行处理

使用 GPU 做并行处理的过程，是 CPU 和 GPU 联动完成的，可以大幅度地提高计算速度。但并不是所有的程序都适合使用 GPU 加速，通常不适合 GPU 加速的有如下两种情况：

1．处理的数据量太小

在数据量很小的情况下仍然采用 GPU 并行处理方式，会因为 CPU 给 GPU 分配工作的时间大于 CPU 自己计算的时间，导致 GPU 加速效果不明显，所以这种情况还不如直接让 CPU 自己运行。好比是网上购买一件产品，邮费的价格已经超过了商品本身的价格，这样的购买肯定不划算。

2．算法本身不适合并行处理

不是所有的算法都适合并行处理，如果算法中后续步骤的计算需要用到之前计算的结果，这种算法就不适合采取并行处理。

给出一个简单例子：现在需要将一个空的一维向量 a[10]填满数据，其中 a[0]＝3，要求后边每个数据的大小是前一个数据的 2 倍。

如果使用串行处理，可以用循环直接写：

$$a[i+1] = a[i] \times 2$$

但是使用 GPU 无法实现这种处理效果，仅使用 GPU 处理的效率会比仅使用 CPU 的效率低很多，增加了处理时间，得不偿失。

在图像处理中，如何使用 GPU 处理图像呢？图像在计算机中存储的样子如图 4-21 所示，像素点以二维矩阵的方式排列并存储。

通常情况下，CPU 会从上至下、从左向右，对每一个像素点做一次计算，如图 4-22 所示。而并行处理，则是先用 CPU 给每个像素点分配一个线程，然后同时对所有的像素点进行处理，最终得到处理结果，如图 4-23 所示。

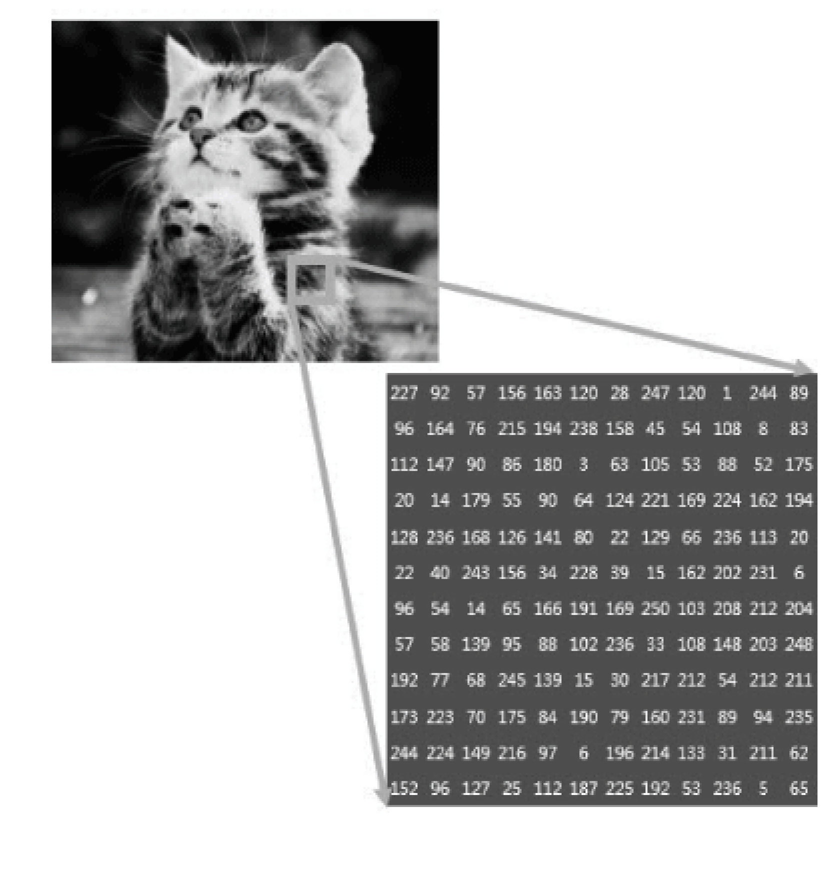


图 4-21　计算机识别图像

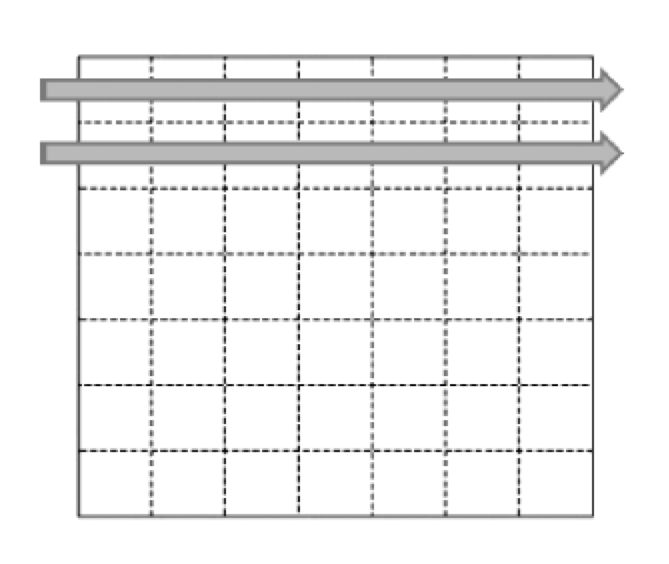


图 4-22　CPU 处理图像数据

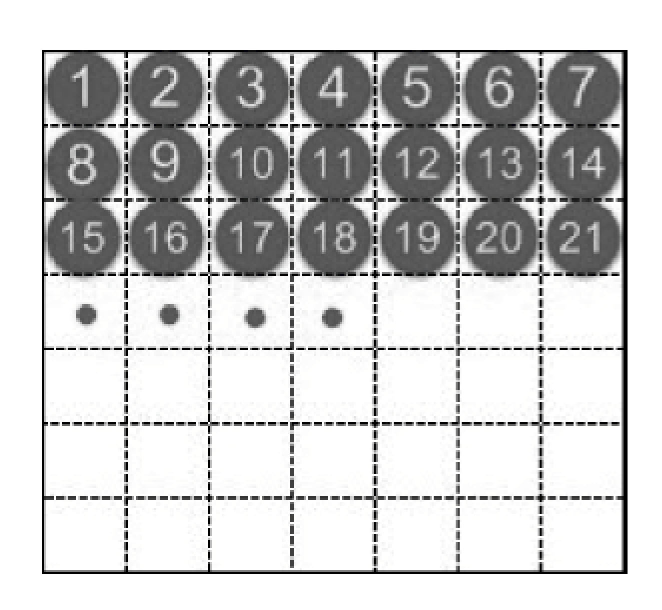


图 4-23　GPU 处理图像数据

4.4　CUDA 环境搭建

4.4.1　CUDA 的下载

可以直接去 NVIDIA 公司的官网下载 CUDA，地址如下：

```
https://developer.nvidia.com/cuda-downloads
```

进入这个网址可以直接看到最新版本的 CUDA 下载选项，如果想下载从前版本的 CUDA，可以在该网页中找到 Documentation 版块，单击 Legacy Toolkits 选项，如图 4-24 所示。

之后进行版本选择，选择需要下载的 CUDA 版本即可，如图 4-25 所示。

Documentation

CUDA Quick Start Guide
Release Notes
EULA
Online Documentation
CUDA Toolkit Overview
Download Checksums
Legacy Toolkits

图 4-24　CUDA 旧版本下载

Latest Release

CUDA Toolkit 7.5 (Sept 2015)

Archived Releases

CUDA Toolkit 7.0 (March2015)
CUDA Toolkit 6.5 (August 2014)
CUDA Toolkit 6.0 (April 2014)
CUDA Toolkit 5.5 (July 2013)
CUDA Toolkit 5.0 (Oct 2012)
CUDA Toolkit 4.2 (April 2012)
CUDA Toolkit 4.1 (Jan 2012)
CUDA Toolkit 4.0 (May 2011)
CUDA Toolkit 3.2 (Nov 2010)
CUDA Toolkit 3.1 (June 2010)
CUDA Toolkit 3.0 (March 2010)
OpenCL 1.0 Release (Sept 2009)
CUDA Toolkit 2.3 (June 2009)
CUDA Toolkit 2.2 (May 2009)
CUDA Toolkit 2.1 (Jan 2009)
CUDA Toolkit 2.0 (Aug 2008)
CUDA Toolkit 1.1 (Dec 2007)
CUDA Toolkit 1.0 (June 2007)

图 4-25　CUDA 旧版本型号

以 CUDA 6.5 为例，找到 CUDA 6.5 一栏，进行计算机系统版本号的选择。如图 4-26 所示，选择 Windows 系统下的 Notebook、64-bit 版本，单击最后的 EXE 即可下载。

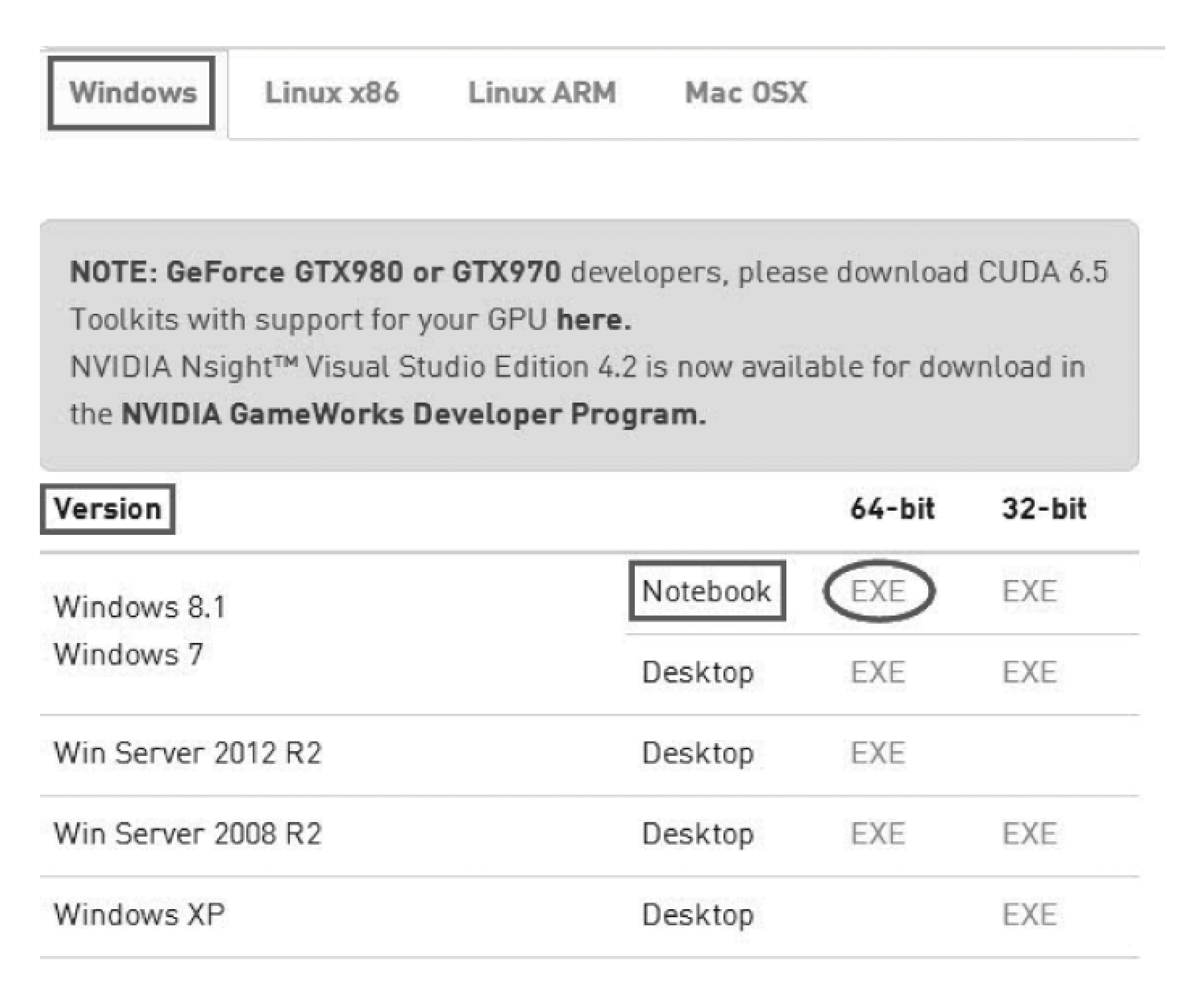


图 4-26　CUDA 版本和系统型号选择

4.4.2　CUDA 的安装

CUDA 的安装相对简单，不过需要注意必须把所有文件都装全，不要仅安装目前所需要的文件。下面以安装 CUDA 6.5 为例进行说明。

(1) 双击安装包，进入安装界面，解压的路径可以有中文，在文件解压完成后会自动进入安装界面，如图 4-27 所示为 CUDA 6.5 安装包，图 4-28 为 CUDA 安装包解压路径。

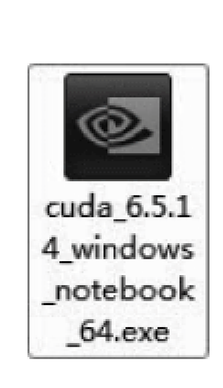


图 4-27　CUDA 6.5 安装包

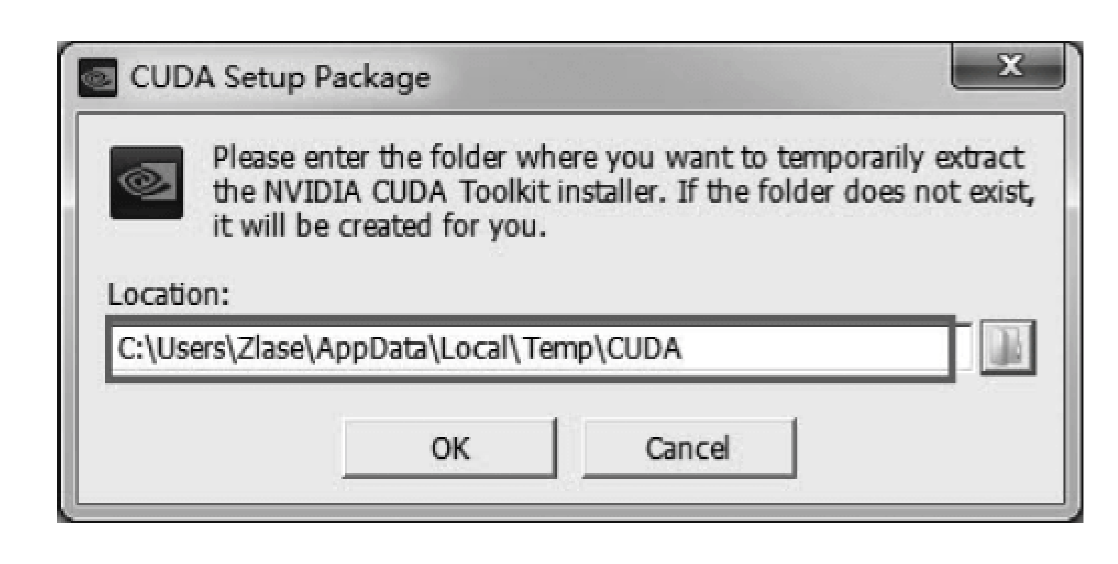


图 4-28　CUDA 6.5 解压路径

(2) 解压成功后会自动进入安装部分，推荐使用 CUDA“自定义(高级)”安装模式，选中所有选项，即安装所有相关文件，如图 4-29 和图 4-30 所示。如果选择“精简(推荐)”模式，可能会导致部分功能无法使用。

(3) 进入下一步，在选项界面将 CUDA 安装在三个文件夹中，如图 4-31 所示。CUDA Tooltik 是 CUDA 的工具包，CUDA Samples 是 CUDA 自带的例程文件，GPU Deployment Kit 则是 GPU 软件开发包，官网介绍它主要是针对 Tesla™、GRID™ 和 Quadro™ 三类显卡



图 4-29　CUDA 安装选项(自定义)



图 4-30　CUDA 安装全部组件

设计的，目前支持 Windows 7、Windows 8、Windows 10 和 Linux 系统。

设定好具体的安装位置后，可以修改文件夹的名字，方便后续添加路径等工作，如图 4-32 所示为安装之后的文件夹目录。

前面提到 CUDA 7.5 无法与 Visual Studio 2015 一同使用，原因是 CUDA 7.5 无法识别 VS 2015，在安装过程中检查兼容性时就能体现出来，如图 4-33 所示，使用 Visual Studio 2015 需要下载最新的 CUDA 8.0。

(4) 安装好 CUDA 后，需要进行环境配置等相关工作。在“我的电脑”的“环境变量”中添加路径，该操作已在 OpenCV 环境配置的部分介绍过。



图 4-31　CUDA 6.5 选择安装位置

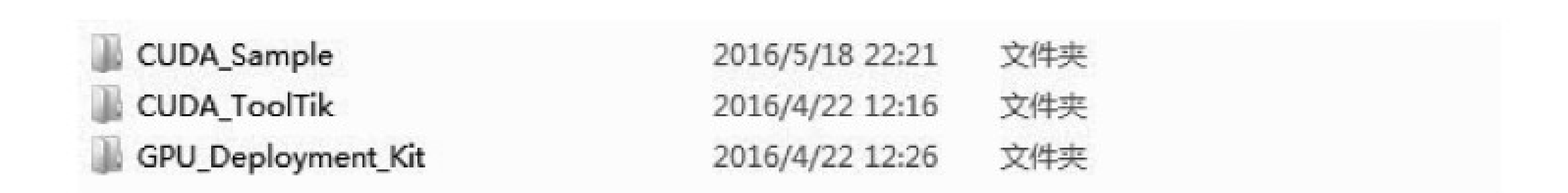


图 4-32　CUDA 安装文件夹

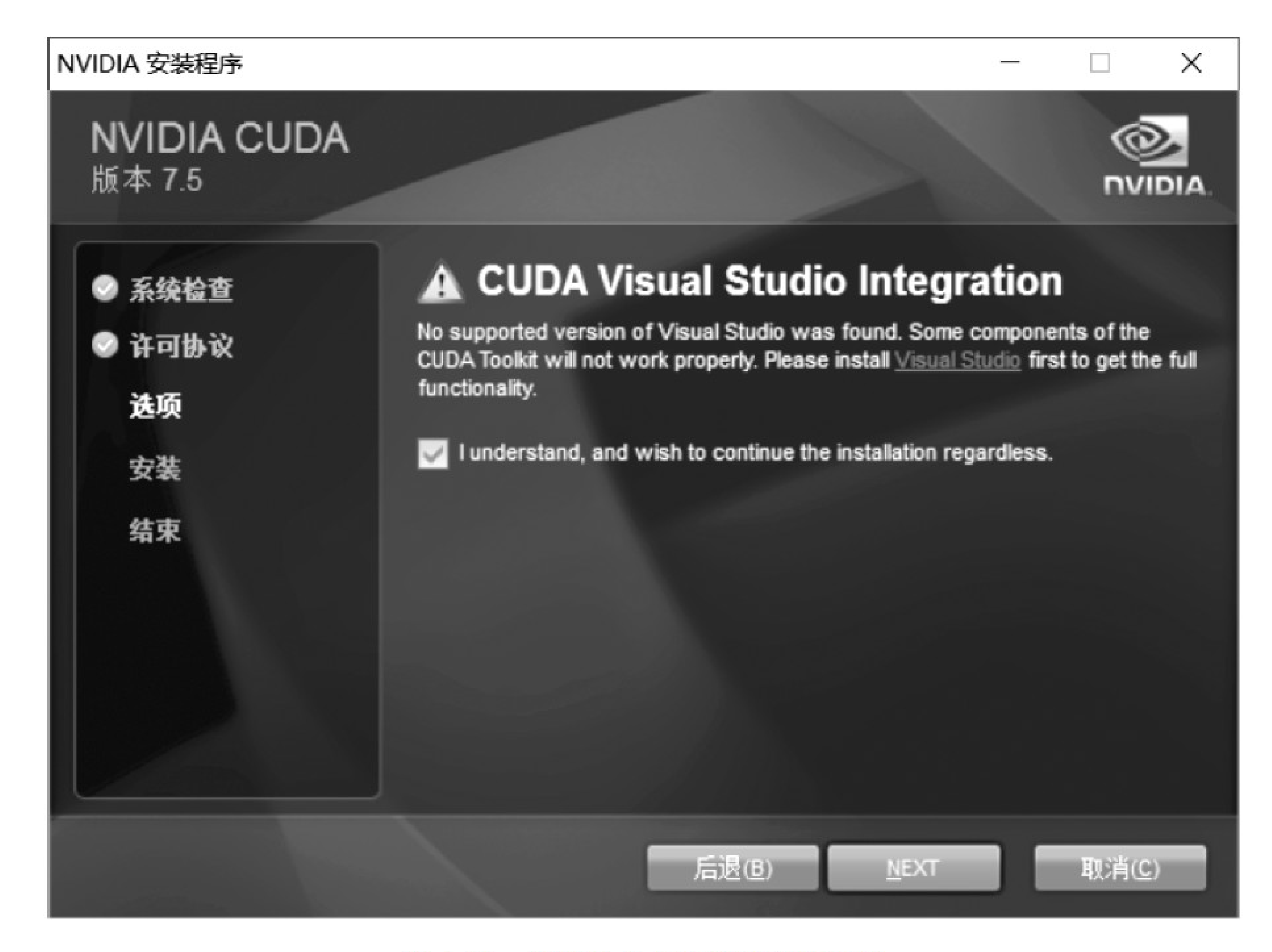


图 4-33　CUDA 7.5 不识别 VS 2015

通常情况下 CUDA 安装好后就默认包含了一些路径，但仍然需要手动在环境变量中添加几条路径，如图 4-34 所示。

```
CUDA_BIN_PATH    D:\CUDA6.5\CUDA_Tooltik\bin
CUDA_LIB_PATH    D:\CUDA6.5\CUDA_ToolTik\lib\win32
CUDA_SDK_BIN     D:\CUDA6.5\CUDA_Sample\bin\win32
CUDA_SDK_LIB     D:\CUDA6.5\CUDA_Sample\common\lib\win32
CUDA_SDK_PATH    D:\CUDA6.5\CUDA_Sample\common
```



图 4-34　CUDA 环境变量路径

(5) 测试。

测试 CUDA 的环境是否搭建成功可以使用 CMD 进行测试。

此处安装路径为 D:\CUDA6.5\CUDA_Sample\bin\win64\Release。

打开 CMD，在上述路径中找到 bandwidthTest.exe 文件，如图 4-35 所示。

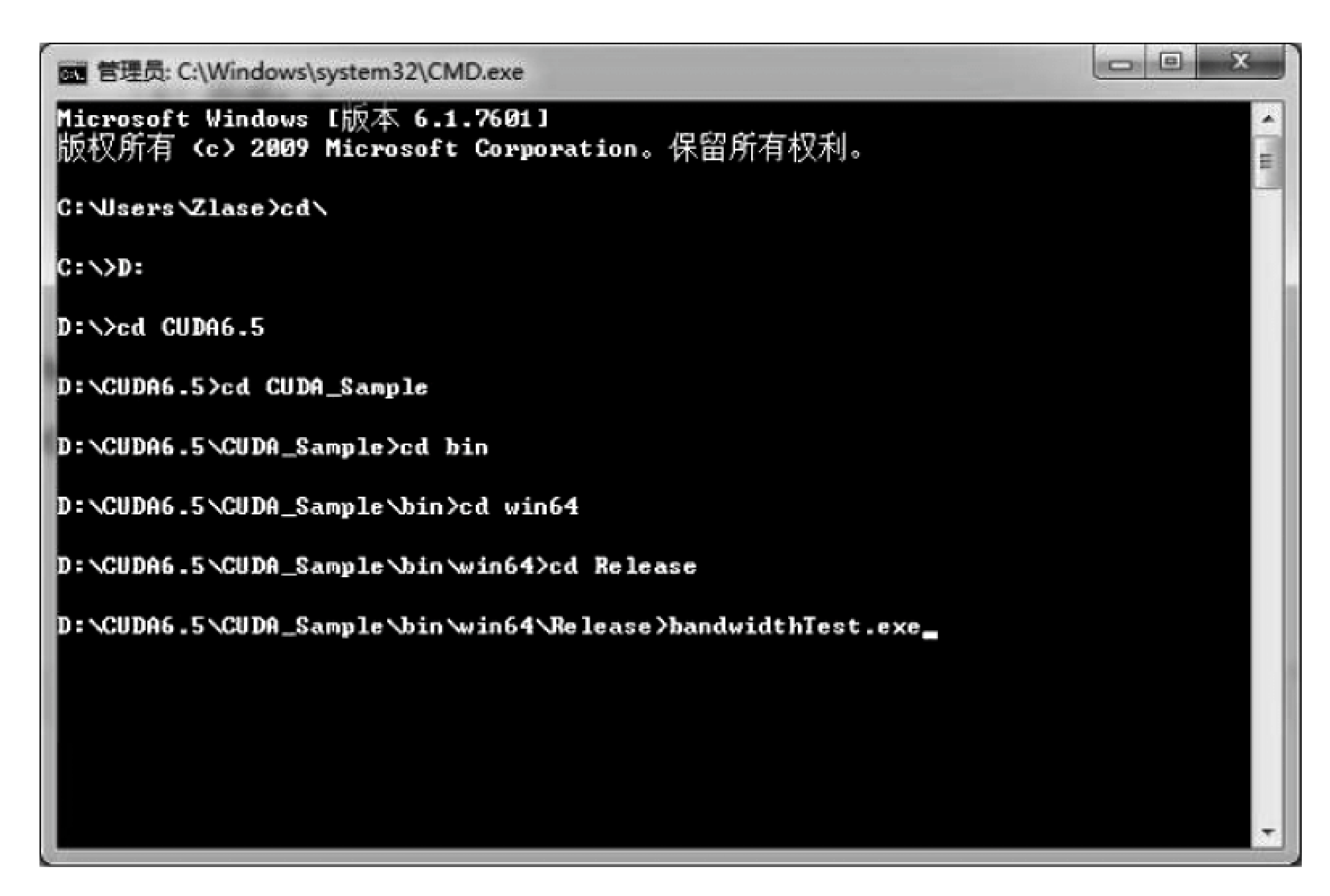


图 4-35　测试 CUDA 是否安装成功

运行 bandwidthTest. exe 文件，查看结果。运行完成后还需要再运行另外一个名为 deviceQuery. exe 的文件，如果两次的结果都是 PASS，如图 4-36 和图 4-37 所示，则说明整个 CUDA 安装成功。

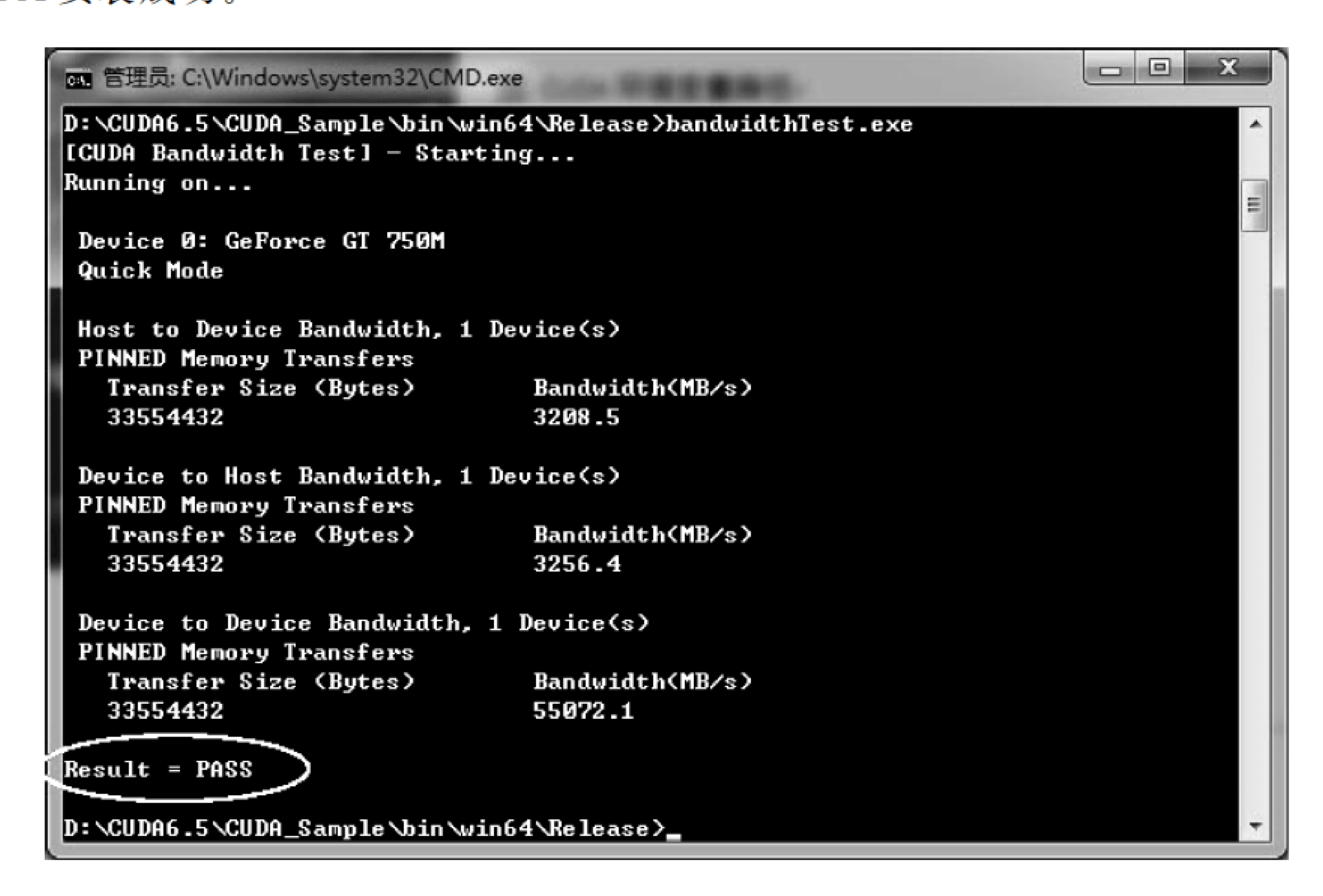


图 4-36 CMD 测试 bandwidthTest. exe

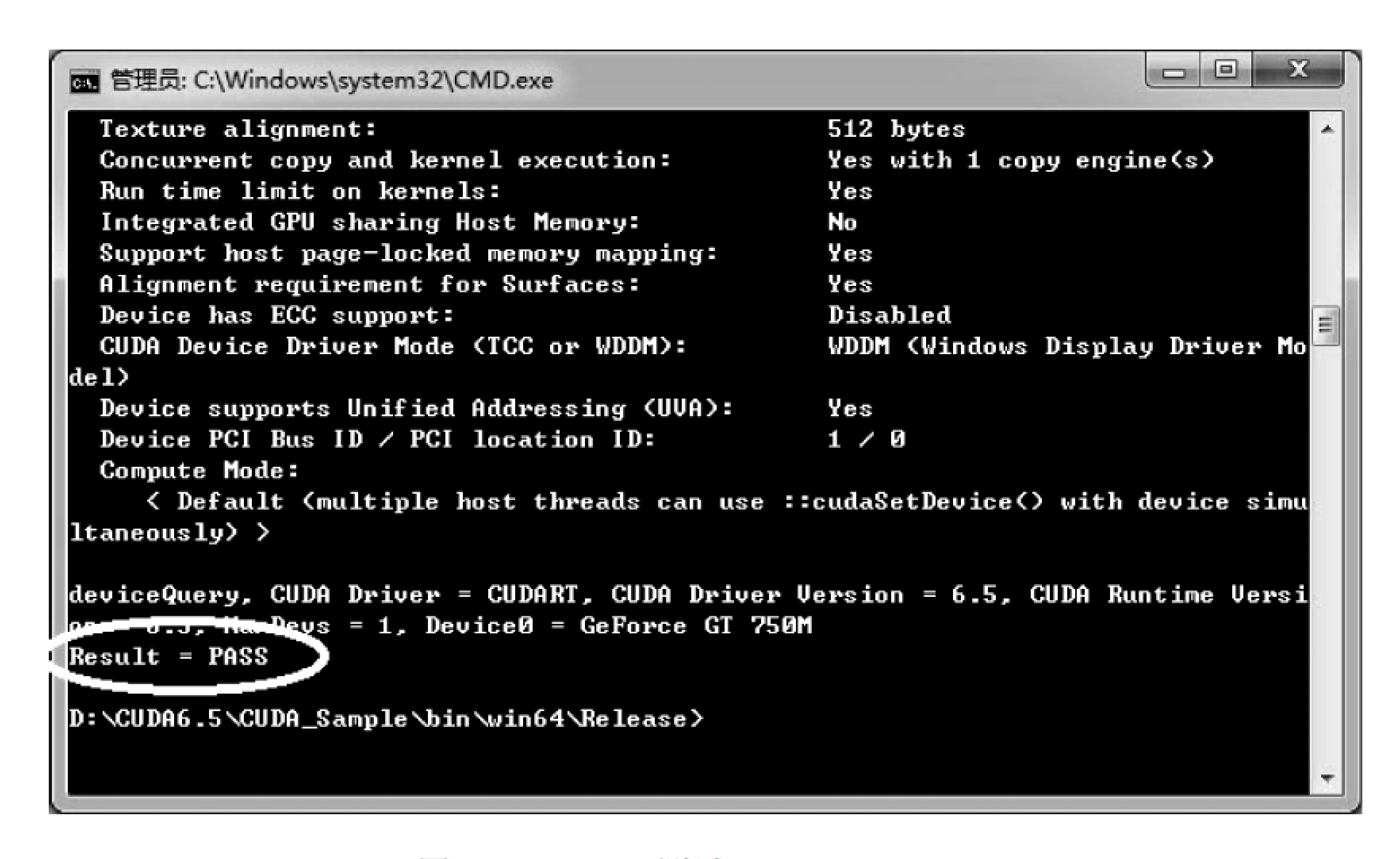


图 4-37 CMD 测试 deviceQuery. exe

4.4.3 CUDA 在 VS 中的测试

安装好 CUDA 后，可以使用 Visual Studio 来进行环境测试。

(1) 在 VS 中新建一个项目文件，在“新建项目”窗口选择 NVIDIA 中的 CUDA 6.5，之后建立这个项目文件即可，如图 4-38 所示。

(2) 建立好项目文件后可以打开其自带的程序，在如图 4-39 所示的位置加上一个





图 4-38 创立 CUDA 项目文件

getchar()防止程序运行结束自动退出。如果成功运行，则说明 CUDA 可以在 VS 平台上使用，即可进行后续开发。

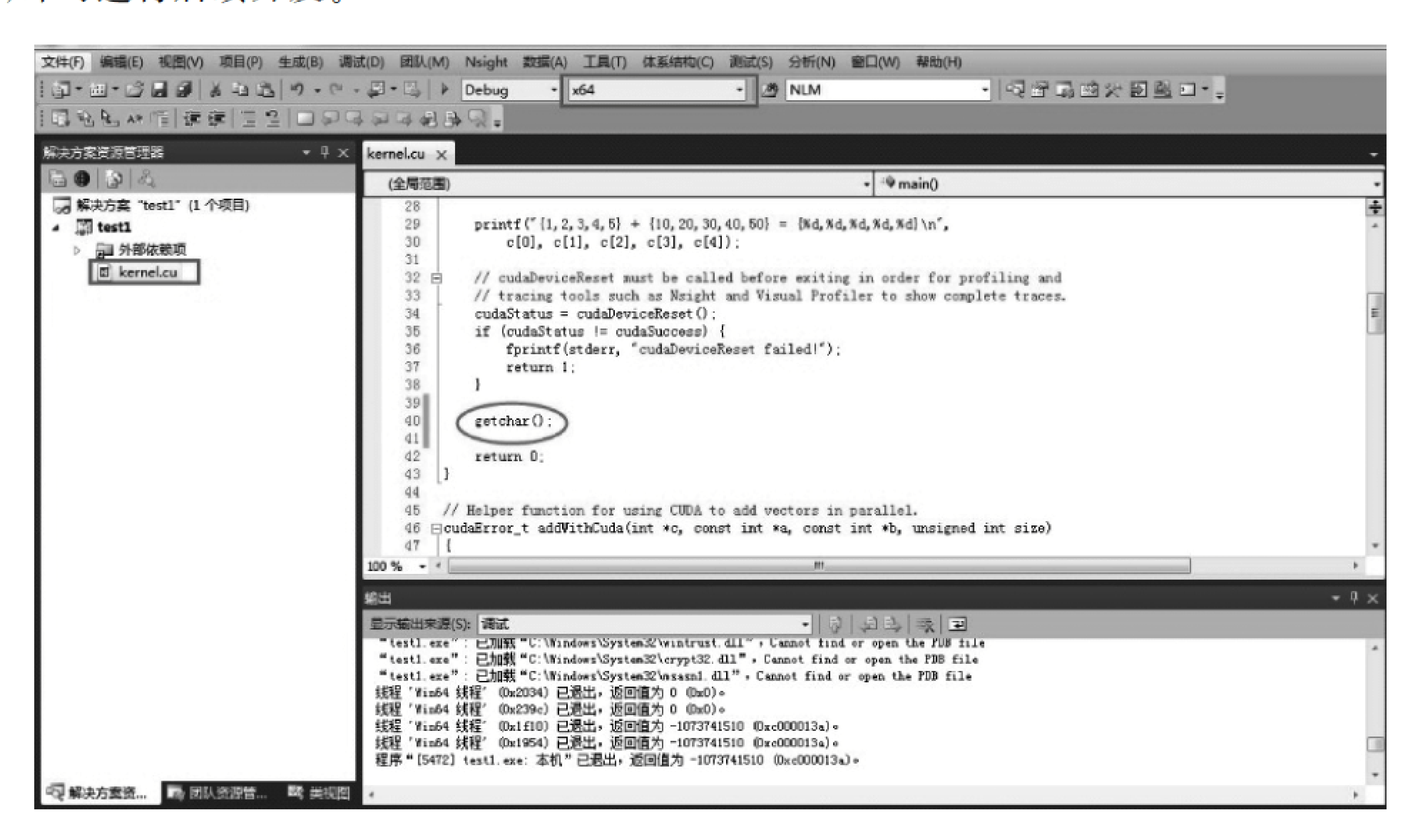


图 4-39 VS 中的测试

4.4.4 CUDA 项目的创建

平时编写的程序跟例子中的模式不太一样，需要将.cpp 文件和.cu 文件分开写，因此必须重新配置项目文件。这种纯手动配置 CUDA 的过程更加烦琐，但是对一些比较大的工程或.cu 文件较多的工程，推荐使用这种配置方式，虽然配置烦琐但条理清晰。接下来将介绍

如何手动创建并配置 CUDA 项目文件。

（1）CUDA 关键字高亮配置。

在正常的.cpp 文件中，一些函数名称、变量会显示为蓝色字体，与其他黑色字体形成对比。但是在编写 CUDA 程序时，需要在.cu 或者.cuh 文件中编写，因此必须将.cu 文件中的 C++函数和部分 CUDA 函数调整为高亮。这一步的配置是 VS 中的配置，因此完成这一步，以后其他 CUDA 程序都不再需要重新进行关键字高亮调整了。

首先是将.cu 文件中的 C++函数高亮。

打开 VS 后，在上边的菜单栏选项中选择“工具”→“选项”命令，在“选项”窗口中选择“文本编译器”→“文件扩展名”选项，如图 4-40 所示。在“扩展名”里分别添加 cu 和 cuh 并在旁边的编辑器下拉菜单中选择 Microsoft Visual C++选项，添加即可。



图 4-40 .cu 文件 C++高亮

第二步实现在,cu 文件中使 CUDA 函数高亮。

如果采用的 CUDA 版本是 CUDA 6.0 或者 CUDA 6.0 以下的版本，可以在

```
SDK_PATH\C\doc\syntax_highlighting\usertype.dat
```

路径下找到 usertype.dat 文件，这个文件中包含的是常用 CUDA 函数、字符定义等。将这个文件复制粘贴到 VS 中的 IDE 文件夹中，重新启动 VS 即可配置完成，本书示例的路径为：

```
D:\VS2010\Common7\IDE
```

如果采用的是 CUDA 6.0 以上的版本，比如本书使用的 CUDA 6.5，则没有这个文件，因为此文件在较高的 CUDA 版本中已经被删除。因此，该文件需要在网上下载或者自己重新写一个。

如何写 usertype. dat 文件呢？

先建立一个. txt 文件，在这个文件中输入想要高亮的 CUDA C 语言的函数、变量等。格式是每输入一个函数或变量，按 Enter 键，然后再输入下一个需要高亮的函数或变量，如图 4-41 所示。



图 4-41　usertype. dat

需要输入的高亮函数的有__global__、__host__、__device__、__constant__、__shared__、gridDim、blockIdx、blockDim、threadIdx、char1、char2、char3、char4、uchar1、uchar2、uchar3、uchar4、short1、short2、short3、short4、ushort1、ushort2、ushort3、ushort4、int1、int2、int3、int4、uint1、uint2、uint3、uint4、long1、long2、long3、long4、ulong1、ulong2、ulong3、ulong4、longlong1、longlong2、float1、float2、float3、float4、double1、double2、dim1、dim2、dim3、dim4、tex1Dfetch、tex1D、tex2D、tex3D、__float_as_int、__int_as_float、__float2int_rn、__float2int_rz、__float2int_ru、__float2int_rd、__float2uint_rn、__float2uint_rz、__float2uint_ru、__float2uint_rd、__int2float_rn、__int2float_rz、__int2float_ru、__int2float_rd、__uint2float_rn、__uint2float_rz、__uint2float_ru、__uint2float_rd、__fadd_rz、__fmul_rz、__fdividef、__mul24、__umul24、__mulhi、__umulhi、__mul64hi、__umul64hi、min、umin、fminf、fmin、max、umax、fmaxf、fmax、abs、fabsf、fabs、rsqrtf、rsqrt、sqrtf、sqrt、__sinf、sinf、sin、__cosf、cosf、cos、__sincosf、sincosf、sincos、__expf、expf、exp、__logf、logf、log、__syncthreads。

之后将这个文件重命名为 usertype. dat，注意后缀也要改。将该文件放在

```
D:\VS2010\Common7\IDE
```

文件夹中即可实现 CUDA 中的函数高亮，高亮函数可以为编程提供很大的便利。

（2）在新建项目中选择 Visual C++模板下的“常规”选项，之后选择“空项目”创立该文件，如图 4-42 所示。

（3）右击项目文件，选择“属性”命令，进入属性界面，如图 4-43 所示，在“VC++目录”中找到右侧的“包含目录”“引用目录”和“库目录”。

其中“包含目录”要将 CUDA 中 ToolTik 的 include 文件夹包含进去，因此示例中添加的路径为：

```
D:\CUDA6.5\CUDA_ToolTik\include
```



图 4-42　建立空项目文件



图 4-43　VC++目录配置

“引用目录”和“库目录”中都要添加的一条相同的路径为：

```
D:\CUDA6.5\CUDA_ToolTik\lib\x64
```

(4) 在“VC++目录”配置完成之后，单击“链接器”再找到“常规”右侧的“附加库目录”这一项，在这中间填上 tooltik 对应的 lib 文件夹，如图 4-44 所示，示例路径为：

```
D:\CUDA6.5\CUDA_ToolTik\lib\x64
```

当然也可以使用在计算机环境变量中的动态路径。如果使用 32 位，那么在 lib 文件夹中可以选择 win32。因为示例中使用的是 64 位，所以填写的都是使用 x64 的 lib 文件。

(5) 配置完附加库目录后，单击“常规”下的“输入”选项，并在其右侧找到“附加依赖项”，如图 4-45 所示，在其中填入 cudart.lib，如图 4-46 所示。

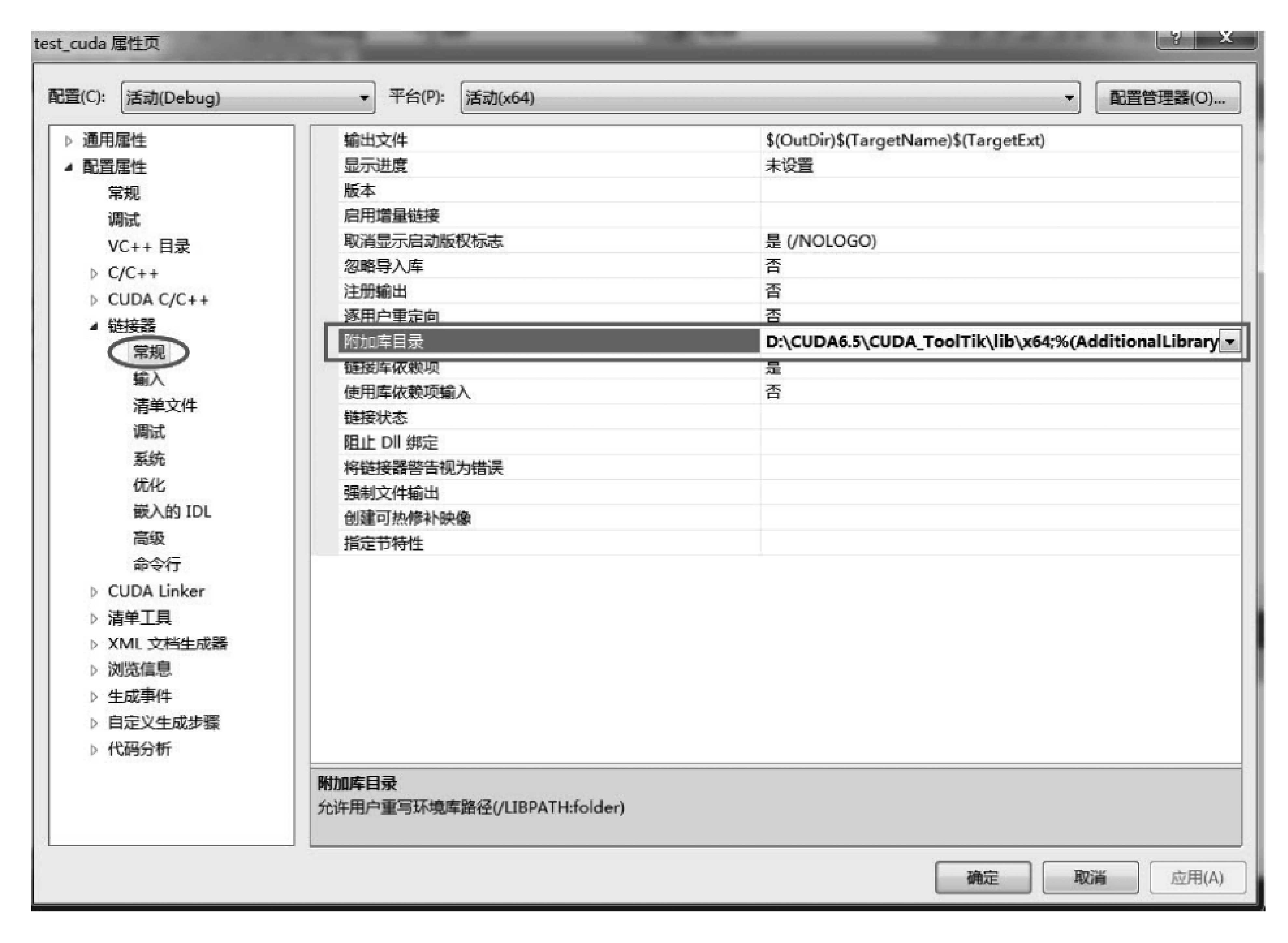


图 4-44　附加库目录配置

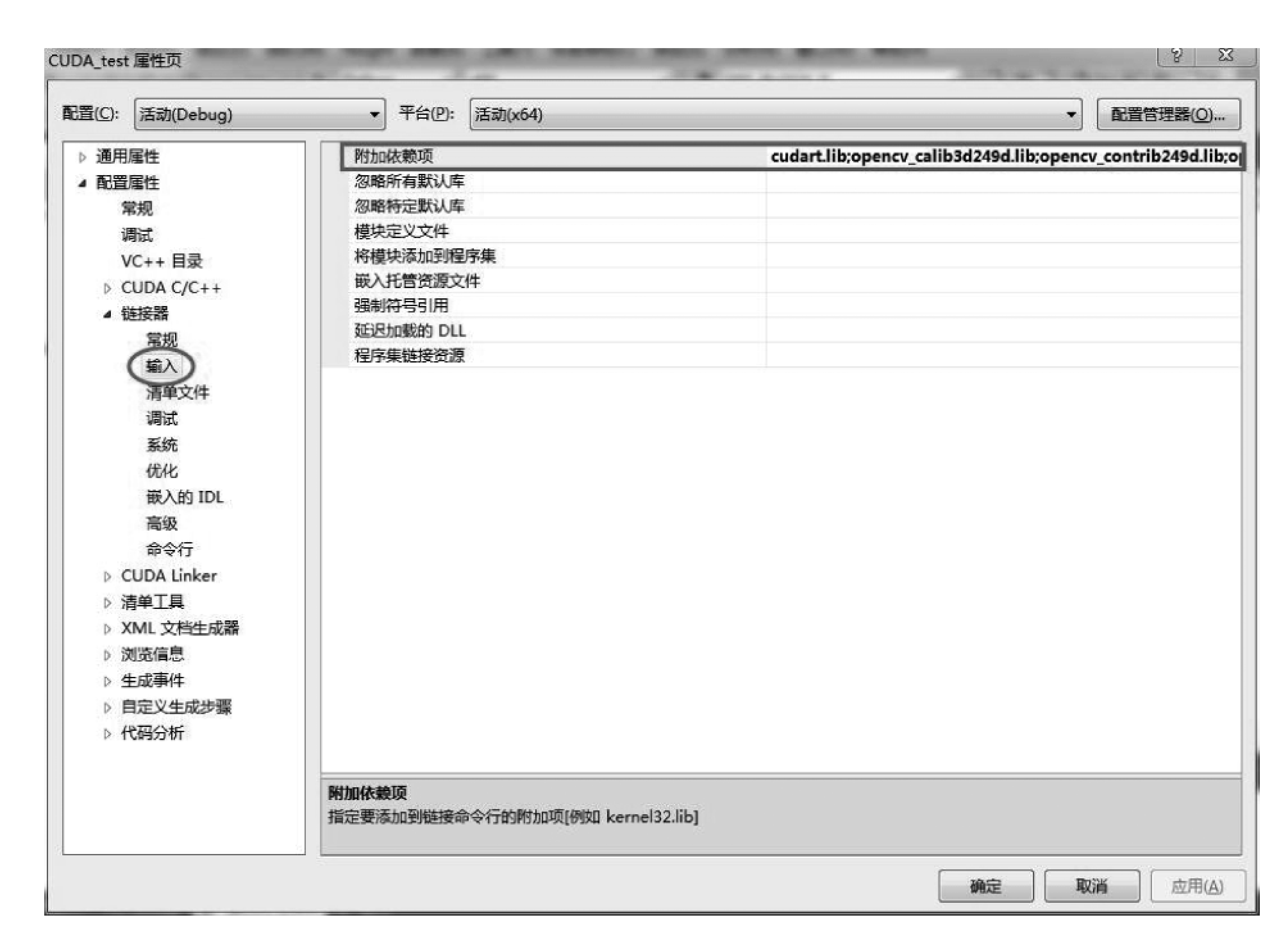


图 4-45　附加依赖项配置第一步



图 4-46　附加依赖项配置第二步

(6) 将属性页左上角的“配置”选项由 Debug 更换成 Release，然后再进行一遍同样的配置，如图 4-47 所示。当然，如果仅仅进行 Debug，就可以不用在 Release 模式下重新配置；但如果程序需要 Release，则必须重新配置。

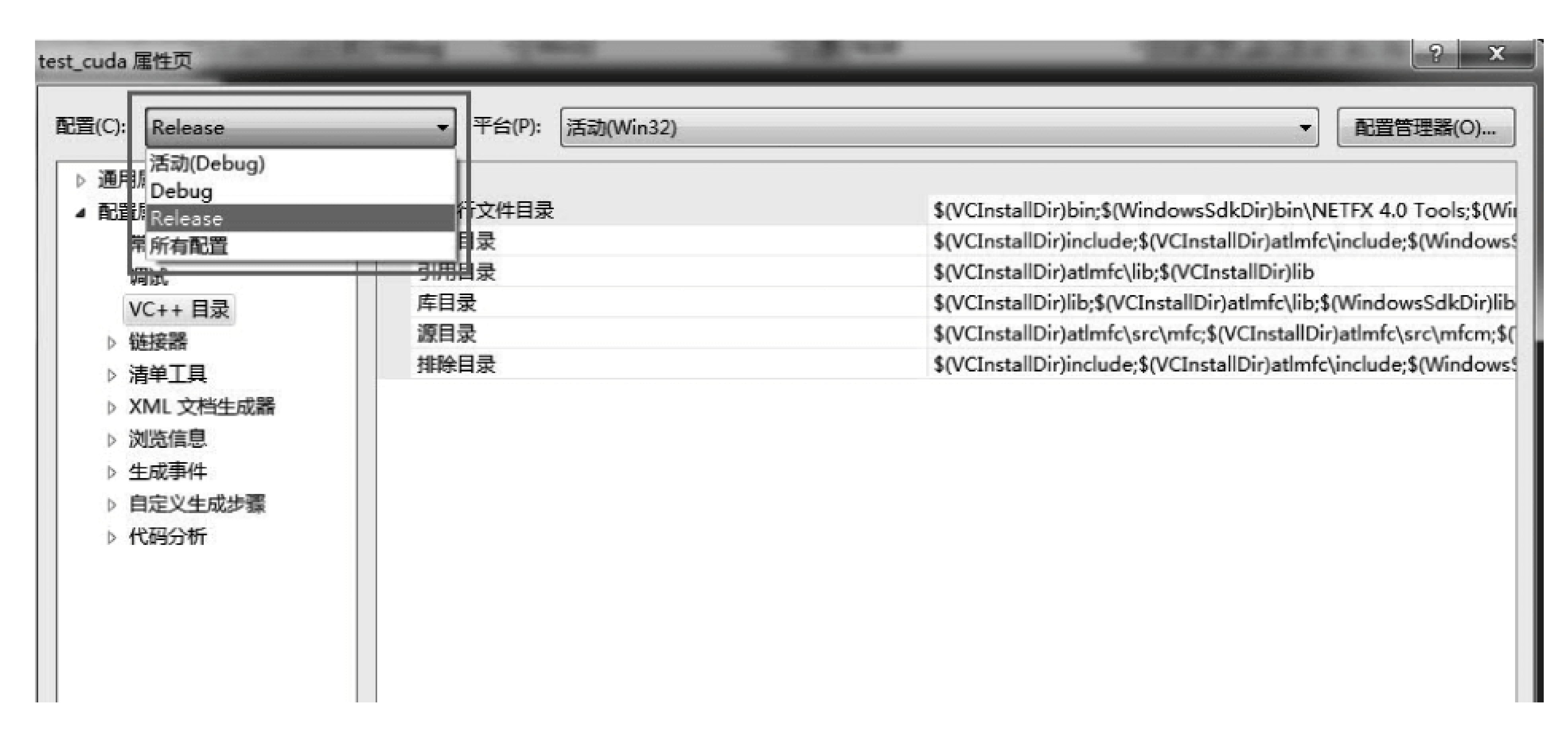


图 4-47　属性更换 Release 配置

(7) 配置环境结束后，即可在“源文件”中添加新建项，如 C++ 的. cpp 文件和 GPU 中运行的. cu 文件，如果需要建立. cuh 文件，直接选择 CUDA C/C++ Header 即可，如图 4-48～图 4-50 所示。

(8) 右击工程文件，选择“生成自定义”命令，然后在出现的窗口选中 CUDA 6.5，如图 4-51 所示。

(9) 右击项目中的. cu 文件，选择“属性”命令，在出现的窗口选择“常规”选项，在右侧的下拉列表框中选择 CUDA C/C++ 选项，如图 4-52 所示。至此，整个 CUDA 文件配置完成。



图 4-48 添加新建项

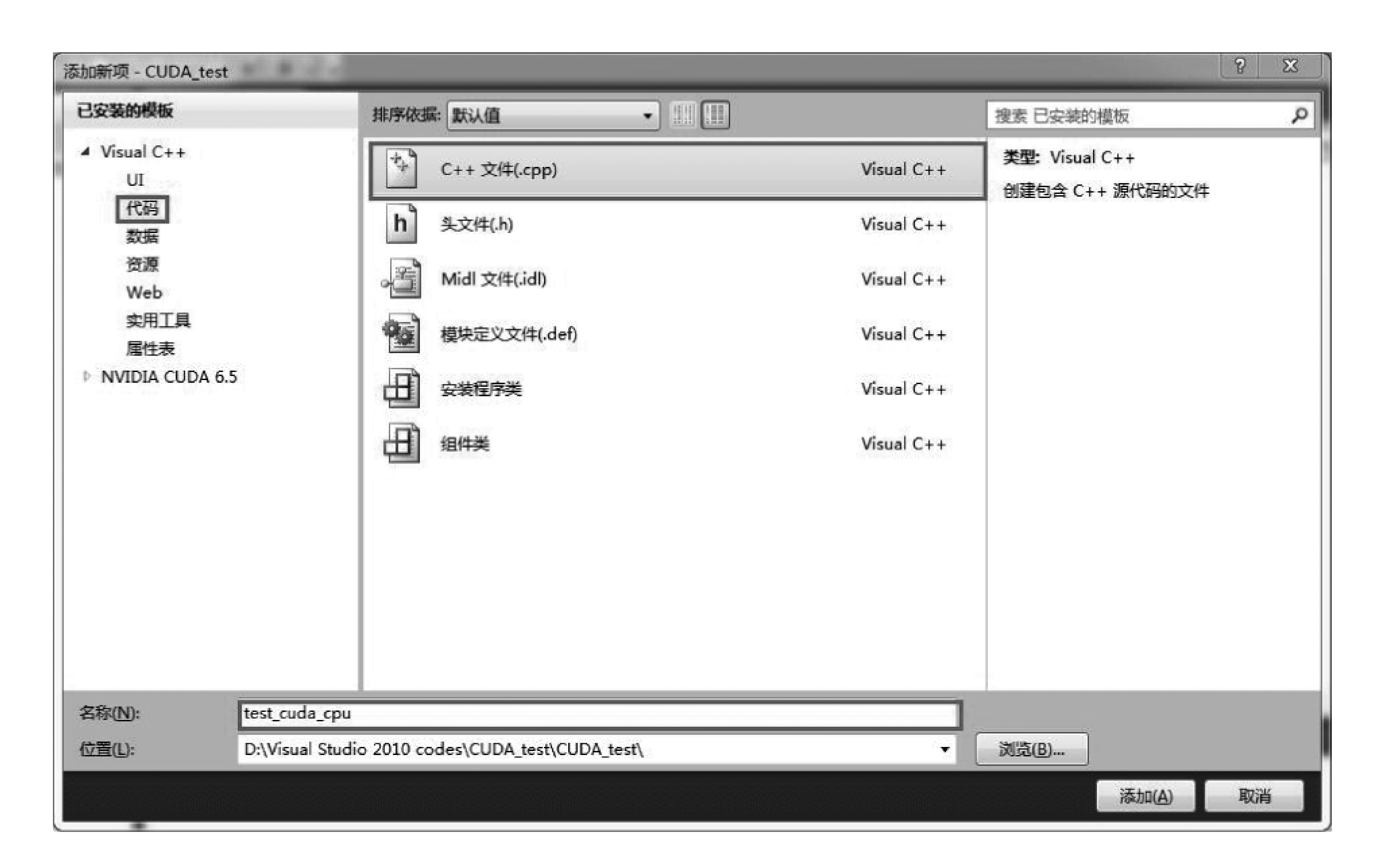


图 4-49 添加.cpp 文件

需要注意的是，在下拉列表框中一定选择 CUDA C/C++而不是“C++编译器”或者其他的选项。

(10) 现在需要对环境进行测试，可以将 4.4.3 节的 kernel.cu 中自带的程序复制到.cu 文件中，直接运行查看环境是否配置成功，如图 4-53 所示为测试的结果。

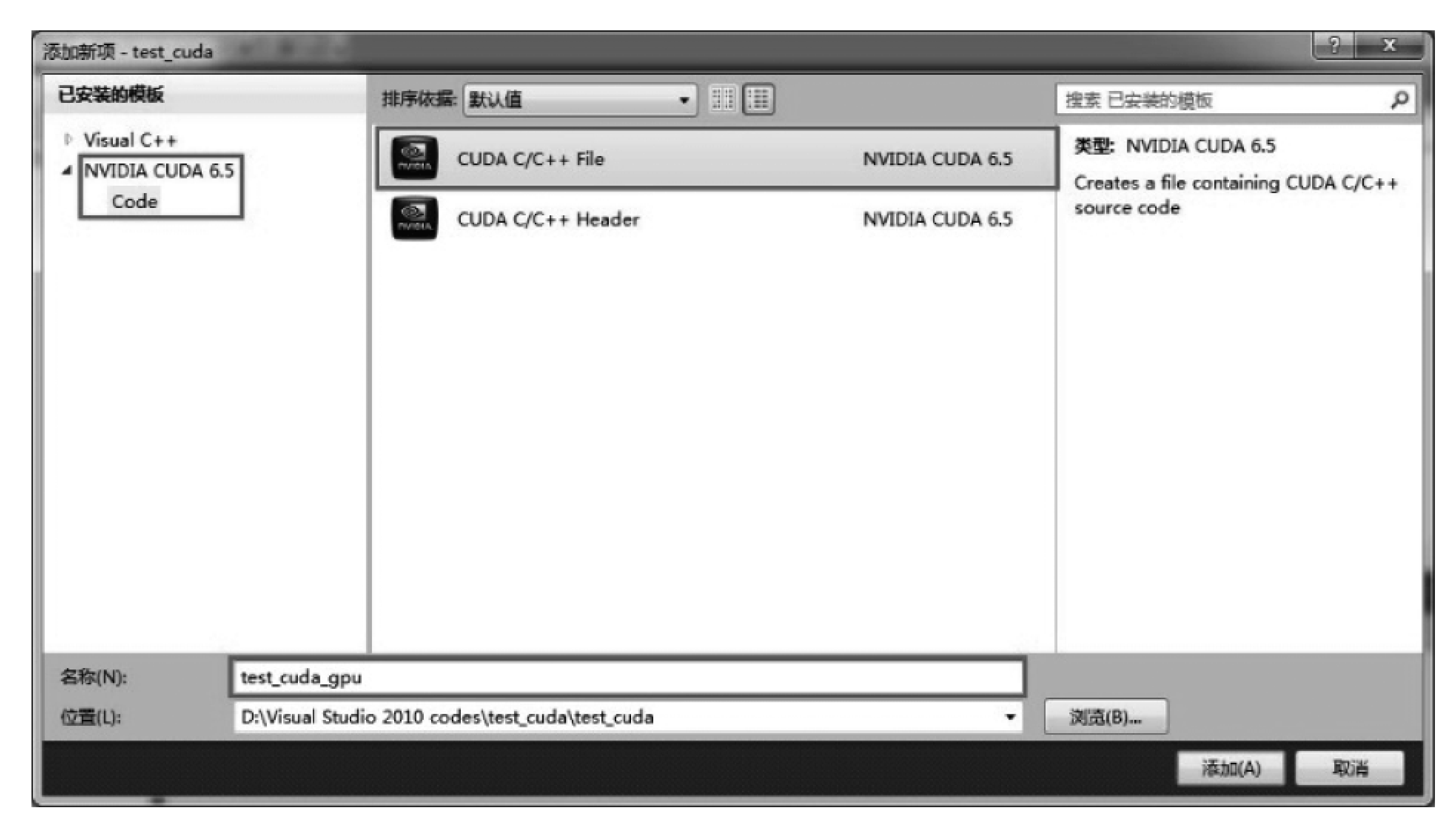


图 4-50 添加. cu 文件

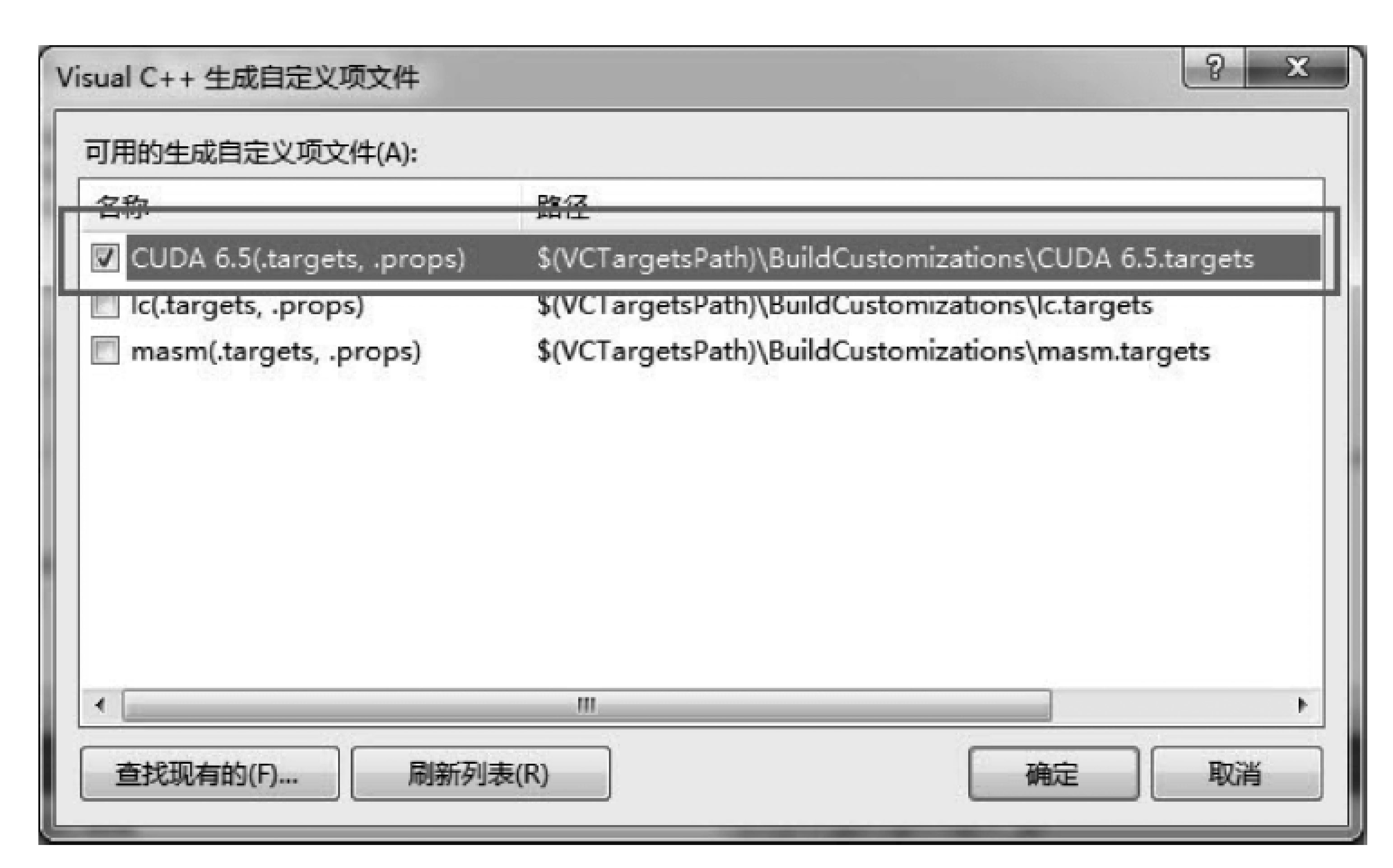


图 4-51 生成自定义



图 4-52 配置. cu 文件

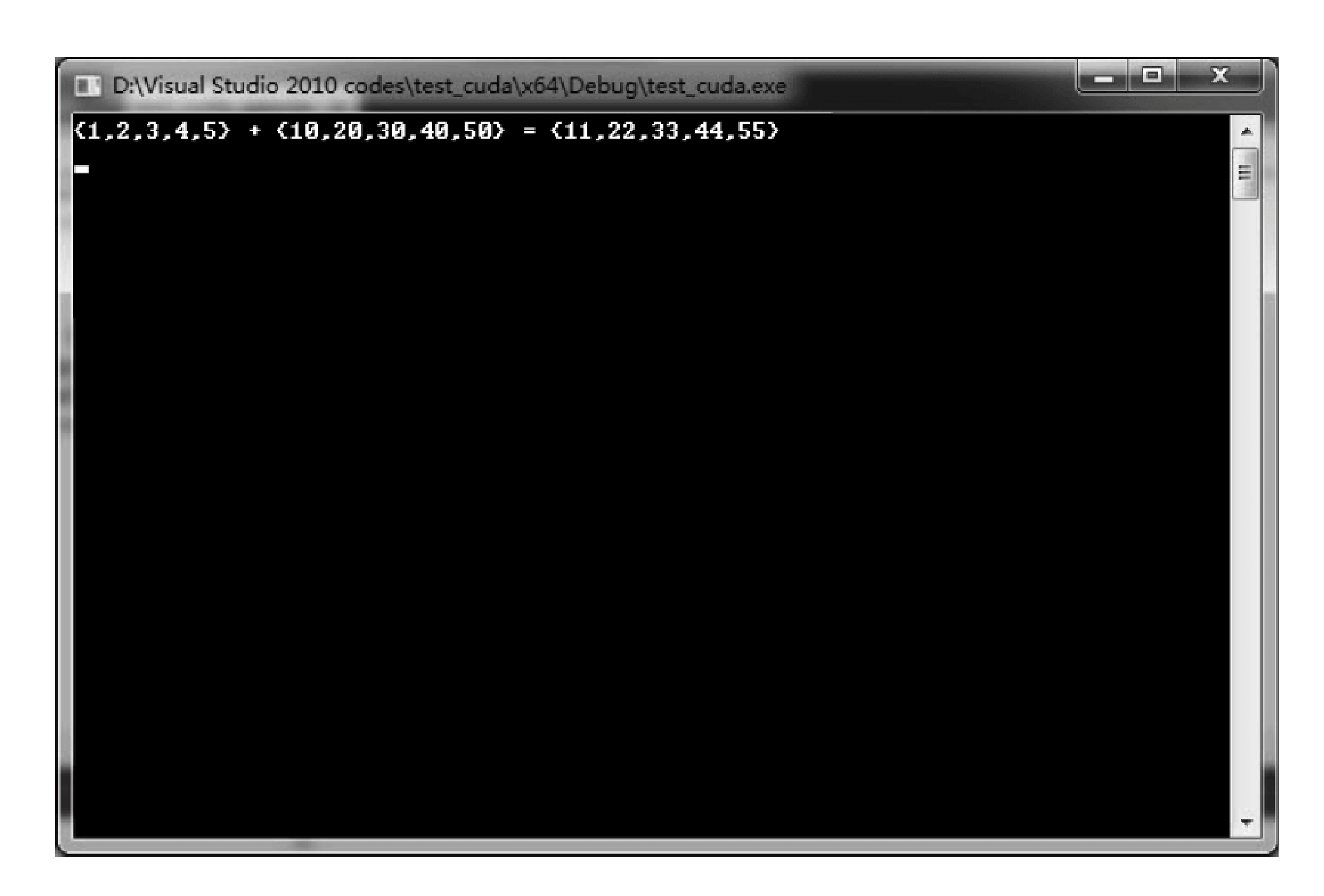


图 4-53　环境配置测试结果

4.5　CUDA C 语言

CUDA C 语言是由 CUDA 程序开发人员提供的一种编写设备端程序的编程语言，原型是 C 语言，并在 C 语言的基础上增加了一些指令，本节将对 CUDA C 语言中的函数进行分类介绍[2]。

4.5.1　C 语言最小扩展集

1. 函数类型限定符

(1) __device__

使用__device__限定符声明的函数在设备上执行，也可通过设备调用。

(2) __global__

使用__global__限定符声明的函数为内核。此类函数在设备上执行，仅可通过主机才能进行调用。

(3) __host__

使用__host__限定符声明的函数在主机上执行，仅可通过主机才能进行调用。

使用过程中需要注意以下问题：

① __device__和__global__函数不支持递归。

② __device__和__global__函数内无法声明静态变量。

③ __device__和__global__函数不得有数量可变的参数。

④ __device__函数的地址无法获取，但支持__global__函数的函数指针。

⑤ __global__和__host__限定符无法一起使用。

⑥ __global__函数的返回型必须为空。

⑦ __global__函数的调用是异步的,也就是说,它会在设备执行完成之前返回。

⑧ __global__函数参数将同时通过共享存储器传递给设备,且限制为256B。

⑨ 若程序中只有__host__声明的函数,则编译器不会将程序分离为主机程序和设备程序,所有的程序均在主机上运行。

2. 变量类型限定符

(1) __device__

__device__限定符声明位于设备上的变量,所声明的变量具有如下属性。

位于:全局存储空间。

生命周期:与应用程序的生命周期一致。

访问方式:可通过网格内的所有线程访问,也可通过运行时库从主机访问。

(2) __constant__

__constant__限定符可选择与__device__限定符一起使用,所声明的变量具有如下属性。

位于:固定存储器空间。

生命周期:与应用程序生命周期一致。

访问方式:可通过网格内的所有线程访问,也可通过运行时库从主机访问。

(3) __shared__

__shared__限定符可选择与__device__限定符一起使用,所声明的变量具有如下属性。

位于:共享存储器空间。

生命周期:与应用程序生命周期一致。

访问方式:可通过网格内的所有线程访问,也可通过运行时库从主机访问。

但是在使用时需要注意以下问题:

① 主机程序中的struct和union成员、形参和局部变量不能使用以上限定符。

② __shared__和__constant__变量具有隐含的静态存储。

③ 以上限定符无法使用extern关键字定义外部变量。

④ __device__和__constant__变量仅允许在文件作用域内使用。

⑤ 声明__shared__变量时不能包含初始化。

⑥ __constant__变量仅可通过主机运行时函数从主机指派。

⑦ 在设备程序中声明、不带任何限定符的自动变量通常位于存储器中。

⑧ 通过__device__、__shared__或__constant__变量的指针获得的地址仅可在设备程序中使用。通过cudaGetSymbolAddress()获取的__device__或__constant__变量地址仅可在主机程序中使用。

3. 调用内核

在CUDA程序中,用__global__函数类型限定符定义的函数称为内核,用＜＜＜　＞＞＞的格式输入需要调用的CUDA线程数来调用内核,表达式具体格式为＜＜＜Dg,Db,Ns,s＞＞＞。例如,需要调用一个内核程序__global__void Func(float * parameter){ },则必须通过如下方法来调用此函数:

```
Func <<< Dg,Db,Ns,s >>>(parameter);
```

其中：

（1）Dg 的类型为 dim3，指定线程网格的维度和大小，Dg. x * Dg. y 等于所启动线程块的数量，Dg. z 设定为 0。

（2）Db 的类型为 dim3，指定线程块的维度和大小，Db. x * Db. y * Db. z 等于每个线程块的线程数量。

（3）Ns 的类型为 size_t，指定为此调用动态分配的共享存储器的大小（除静态分配的存储器），这些动态分配的存储器可供声明为 extern 的数组使用，Ns 是一个可选参数，默认值为 0。

（4）s 的类型为 cudaStream_t，指定相关流；s 是一个可选参数，默认值为 0。

执行配置的参数将在实际函数参与之前被评估，与函数参数相同，通过共享存储器同时传递给设备。

如果 Dg 或 Db 大于设备允许的最大线程数量或 Ns 大于设备上可用的共享存储器最大值，或小于静态分配、函数参数和执行配置所需的共享存储器数量，函数将报错。

4. 内置变量

（1）gridDim：线程网格的维度，类型为 dim3。

（2）blockIdx：线程网格内的线程块索引，类型为 uint3。

（3）blockDim：线程块的维度，类型为 dim3。

（4）threadIdx：线程块内的线程索引，类型为 uint3。

（5）warpSize：线程为单位的 warp 块的大小，类型为 int。

需要注意的是，程序中的变量不允许接受任何内置变量地址，也不允许为任何内置变量赋值。

5. 内置变量类型

（1）char1、uchar1、char2、uchar2、char3、uchar3、char4、uchar4、short1、ushort1、short2、ushort2、short3、ushort3、short4、ushort4、int1、uint1、int2、uint2、int3、uint3、int4、uint4、long1、ulong1、long2、ulong2、long3、ulong3、long4、ulong4、float1、float2、float3、float4、double2。

这些向量类型继承自基本整型和浮点类型。它们均为结构体，第 1、2、3、4 组件分别可通过字段 x、y、z 和 w 访问。它们附带形式为 make_< type name >的构造函数，示例如下：

```
int2 make_int2(ont x,int y);
```

（2）dim3 类型。此类型是一种整型向量类型，基于用于指定维度的 uint3。在定义类型为 dim3 的变量时，未指定的任何组件都将初始化为 1。

6. 计时函数

```
clock_t clock();
```

在设备程序中执行时，返回随每一次时钟周期而递增的每个多处理计数器的值。在内核启动和结束时对此计数器取样，确定两次取样的差别，然后为每个线程记录下结果。这为各线程提供了一种度量方法，可度量设备为了完全执行线程而占用的时钟周期，但不是设备在执行线程指令时实际使用的时钟周期数。



7. 同步函数

__syncthreads()实现线程块内的线程同步,它保证块内的所有线程都执行到该语句的位置。也就是说,执行到该语句的线程会暂停,直到整个块内的线程都执行完成后,才会执行下一步的程序,这样就可以避免产生访存冲突或得到非预期的结果。通常在读写存储器之后,需要调用__syncthreads()函数保证线程同步,但只能在设备端调用。

8. 内存栅栏函数

内存栅栏(memory fence)函数用来保证执行函数的线程所产生的数据能够安全地被其他线程消费,但只能在设备端调用。

内核栅栏函数包括__threadfence()、__threadfence_block()和__threadfence_system()。线程调用__threadfence()函数,其实在该语句之前线程就已经完成对全局存储器或共享存储器的访问,结果对网格内所有线程可见。__threadfence_block()函数的不同在于结果对块内所有线程可见,而线程调用__threadfence_system()函数,结果不仅对 GPU 线程可见,对能够访问主机端页锁定内存的 GPU 线程同样可见,但是只有计算能力 2.0 以上的设备才支持这一特性。

9. 原子函数

原子函数用于在多个线程同时访问全局存储器或共享存储器的同一位置时,实现线程间的互斥操作。各种硬件对原子操作的支持不尽相同,从计算能力 1.1 的设备开始支持全局存储器上的 32 位原子函数;计算能力 1.2 的设备增加了共享存储器上 32 位的原子函数和全局存储器上 64 位的原子函数;计算能力 2.0 以上的设备可以支持共享存储器上 64 位的原子函数。另外,只有原子函数 atomicExch()和 atomicAdd()支持 32 位浮点数的原子操作,其他函数都只支持整型数的原子操作。

10. 纹理函数

CUDA 提供了针对线性内存和 CUDA 数组的纹理拾取(texture fetch)函数。Tex1Dfench()用来从绑定到纹理参考的线性存储器中获取数据。Tex1D()、Tex2D()、Tex3D()分别用于从绑定到纹理参考的一维、二维、三维的 CUDA 数据中获取数据。

4.5.2 运行时库

运行时库是一种被编译器用来实现编程语言的内置函数,以提供该语言程序运行时支持的一种特殊计算机程序库。这种库一般包括基本的输入输出或内存管理等支持。运行时库是程序在运行时所需要的库文件,通常情况下,运行时库是以 lib 或 dll 形式提供的。CUDA 运行时 API 是通过 cudart 动态库提供的,其所有入口都带有 cuda 前缀。cudart 动态库在 CUDA 开发环境安装及配置成功后,在相应安装目录下的 lib 和 bin 文件夹下可以查看到对应 CUDA 版本的 cudart.lib 和 cudart.dll。

CUDA 运行时,API 提供了以下功能的函数:

① 设备管理;

② 线程管理;

③ 流管理;

④ 事件管理;

⑤ 执行管理;

⑥ 存储器管理；

⑦ 纹理引用管理；

⑧ 图形学互操作性；

⑨ OpenGL 互操作性；

⑩ Direct3D 互操作性。

这里列出了所有相关的运行时函数，并简要介绍这些 API 函数的功能和方法，为后续学习打下基础。下面对这些 API 管理函数逐一进行讲解。

1. 设备管理

设备管理函数主要用来选择用于 GPU 计算的设备，枚举所有设备并检索其属性。与设备管理相关的有以下几个函数：

（1）cudaGetDeviceCount

函数功能：返回具有计算能力的设备的数量。

调用形式：cudaError_t cudaGetDeviceCount(int * count)

函数说明：以 * count 形式返回可用于执行的计算能力大于等于 1.0 的设备数量。如果不存在此类设备，cudaGetDeviceCount()将返回 1，且设备 0 仅支持设备模拟模式。由于此设备能够模拟所有硬件特性，所以该设备将报告 9999 中主要和次要计算能力。

返回值：cudaSuccess

（2）cudaSetDevice

函数功能：设置设备以供 GPU 执行使用。

调用形式：cudaError_t cudaSetDevice(int dev)

函数说明：将 dev 记录为活动主线程将执行设备码的设备。

返回值：cudaSuccess；cudaErrorInvalidDevice

（3）cudaGetDevice

函数功能：返回当前使用的设备。

调用形式：cudaReeoe_t cudaGetDevice(int * dev)

函数说明：以 * dev 形式返回活动主线程执行设备码的设备。

返回值：cudaSuccess

（4）cudaGetDeviceProperties

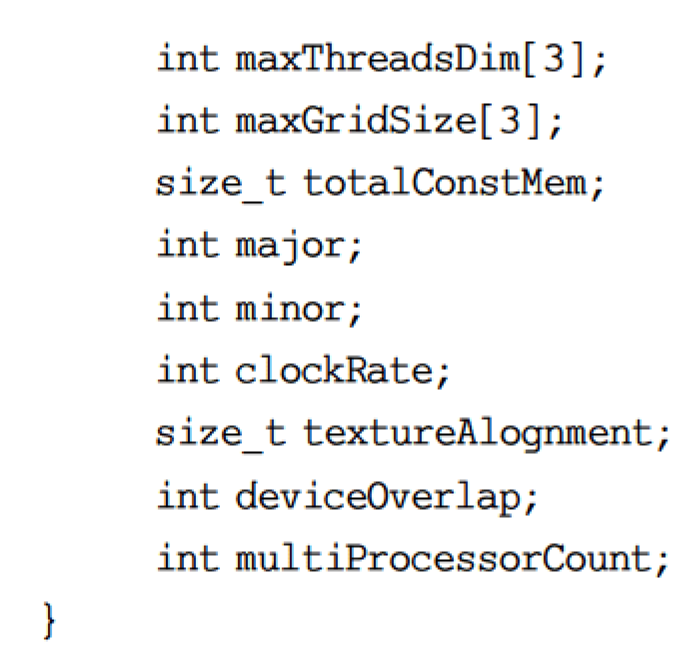
函数功能：返回关于计算设备的信息。

调用形式：cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp * prop,int dev)

函数说明：以 * prop 形式返回设备 dev 的属性。

cudaDeviceProp 结构定义如下：

```
struct cudaDeviceProp{
      char name[256];
      size_t totalGlobalMem;
      size_t sharedMemPerBlock;
      int regsPerBlock;
      int warpSize;
      size_t memPitch;
      int maxThreadsPerBlock;
```



```
        int maxThreadsDim[3];
        int maxGridSize[3];
        size_t totalConstMem;
        int major;
        int minor;
        int clockRate;
        size_t textureAlognment;
        int deviceOverlap;
        int multiProcessorCount;
}
```

其中，各参数含义如下：

name——用于标识设备的 ASCII 字符串。

totalGlobalMem——设备上可用的全局存储器的总量，以字节为单位。

sharedMemPerBlock——线程块可以使用的共享存储器的最大值，以字节为单位；多处理器上的所有线程块可以同时共享这些存储器。

regsPerBlock——线程块可以使用 32 位存储器的最大值；多处理器上的所有线程块可以同时共享这些寄存器。

warpSize——按线程计算的 warp 块的大小。

memPitch——允许通过 cudaMallocPitch()为包含存储器区域的存储器复制函数分配的最大间距，以字节为单位。

maxThreadsDim[3]——块各个维度的最大值。

maxGridSize[3]——网格各个维度的最大值。

totalConstMem——设备上可用的不变存储器总量，以字节为单位。

major 和 minor——定义设备计算能力的主要修订号和次要修订号。

clockRate——以千赫为单位的时钟频率。

textureAlignment——对齐要求，与 textureAlignment 字节对齐的纹理基址不需要对纹理取样应用偏移。

deviceOverlap——如果设备可在主机和设备之间并发复制存储器，同时又能执行内核，则此值为 1；否则为 0。

multiProcessorCount——设备上多处理器的数量。

返回值：cudaSuccess 或 cudaErrorInvalidDevice

(5) cudaChooseDevice

函数功能：选择最匹配标准的计算设备。

调用形式：cudaError_t cudaChooseDevice(int * dev, const struct cudaDeviceProp * prop)

函数说明：以 * dev 的形式返回与 * prop 的匹配程度最高的设备。

返回值：cudaSuccess cudaErrorInvalid Value

例如，查询机器上设备的个数并检索这些设备属性的程序如下：

```
Int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for(device = 0; device < deviceCount;device++)
```

```
{
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp,device);
}
```

需要注意的是,在启动 GPU 计算之前,必须首先调用 cudaSetDevice(device)选择设备,之后才能调用__global__函数或任何来自运行时 API 的函数。如果未通过显式调用 cudaSetDevice()完成此任务,则将自动选中设备 0,随后对 cudaSetDevice()的任何显式调用都将无效。

2. 线程管理

线程管理主要用于管理 GPU 计算线程和 CPU 线程同步问题以及从 CUDA 启动中退出时,清理所有与主机调用线程相关和运行相关的资源。与线程管理相关的函数有以下几个:

(1) cudaThreadSynchronize

函数功能:等待计算设备完成。

调用形式:cudaError_t cudaThreadSynchronize(void)

函数说明:在设备完成所有先前请求的任务之前,一直阻塞操作。如果之前的任务失败,cudaThreadSynchronize()将返回一个错误。

返回值:cudaSuccess

cudaThread Synchronize()实现 GPU 与 CPU 线程的同步。内核函数的启动执行与 CPU 线程执行是异步的,CPU 线程得到的结果未必是内核函数完成之后的结果。如果 CPU 线程需要等待 GPU 线程的执行结果,就必须在内核函数之后调用 cudaThreadSynchronize()函数,让 CPU 线程等待 GPU 线程退出。类似的函数还有 cudaStreamSynchronize()和 cudaEventSynchronize(),它们用于流和事件的同步。从主机测量一系列 CUDA 调用所需时间时,要首先调用 cudaThreadSynchronize()函数等,使 GPU 线程执行完毕后,进入 CPU 线程,从而得到正确的测时。

(2) cudaThreadExit

函数功能:从 CUDA 启动中退出并清除。

调用形式:cudaError_t cudaThreadExit(void)

函数说明:显式清除与调用主线程有关的相关资源,后续任何 API 调用都将重新初始化运行时;在主线程退出时,将隐式调用 cudaThreadExit()。

返回值:cudaSuccess

3. 流管理

流管理函数主要包含了流的创建、销毁,查询流的状态和流的同步问题。流主要用于管理 CUDA 函数异步执行时的并发问题。与流管理相关的函数有以下几个:

(1) cudaStreamCreate

函数功能:创建异步流。

调用形式:cudaError_t cudaStreamCreate(cudaStream_t * stream)

函数说明:创建流。

返回值:cudaSuccess; cudaErrorInvalid Value

（2）cudaStreamQuery

函数功能：查询流完成的状态。

调用形式：cudaError_t cudaStreamQuery(cudaStream_t stream)

函数说明：如果流中的所有操作均已完成，则返回 cudasuccess，否则返回 cudaErrorNotReady。

返回值：cudaSuccess；cudaErrorN0tReady；cudaErrorInvalidResourceHandIe

（3）cudaStreamSynchronize

函数功能：等待流任务完成。

调用形式：cudaError_t cudaStreamSynchronize(cudaStream_t stream)

函数说明：在设备完成流中的所有操作之前，一直阻塞操作。

返回值：cudaSuccess；cudaErrorInvalidResourceHandle

（4）cudaStreamDestroy

函数功能：销毁并清除流对象。

调用形式：cudaError_t cudaStreamDestroy(cudaStream_t stream)

函数说明：销毁流对象。

返回值：cudaSuccess；cudaErrorInvalidResourceHandle

例如，创建两个流，并为每个流定义一个序列，包括一次从主机到设备的存储器复制、一次内核启动和一次从设备到主机的存储器复制。程序如下：

```
cudaStream_t stream[2];                    //创建两个流
for (int i = 0; i < 2; i++)
    cudaStreamCreate (&stream [i]);        //为每个流定义一个序列
for (int i = 0; i < 2; i++)
    cudaMemcpyAsync (inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice,
stream[i] );
for (int i = 0; i < 2; i++)
    kernel <<< 100, 512, 0, stream[i]>>>
      (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; i++)
     cudaMemcpyAsync (hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost,
stream[i]);
       cudaThreadSynchronize();
```

在上面的程序中，两个流均会将输入数组 hostPtr 的一部分复制到设备存储器的 inputDevPtr 数组中，通过调用 myKernel()处理设备上的 inputDevPtr，并将结果 outputDevPtr 复制回 hostPtr。使用两个流处理 hostPtr 允许一个流的存储器复制与另外一个流的内核执行相互重叠。hostPtr 必须指向分页锁定的主机存储器，这样才能出现重叠。

最后调用了 cudaThreadSynchronize()，目的是在进行进一步处理之前确定所有流均已完成。Cudastreamsynchronize()可用于同步主机与特定流，允许其他流继续在该设备上执行。通过调用 cudaStreamDestroy()可释放流。

4. 事件管理

事件管理包括了创建事件对象、记录事件、查询事件、事件同步、销毁事件对象和计算两次事件之间相差的时间这些功能。与事件管理相关的函数有以下几个：

（1）cudaEventCreate

函数功能：创建事件对象。

调用形式：cudaError_t cudaEventCreate(cudaEvent_t * event)

函数说明：创建事件对象。

返回值：cudaSuccess；cudaErrorInitializationError；cudaErrorPriorLaunchFailure；cudaErrorInvalidValue；cudaErrorMemoryAllocation

(2) cudaEventRecord

函数功能：记录事件。

调用形式：cudaError_t cudaEventRecord(cudaEvent_t event，CUstream stream)

函数说明：记录事件。如果流非零，则在完成流中所有先前操作之后记录事件；否则将在完成 CUDA 上下文中所有先前的操作之后记录事件。由于此操作是异步的，所以必须使用 cudaEventQuery()或 cudaEventSynchronize()确定事件的实际记录时间。如果之前已调用过 cudaEventRecord()，而尚未记录事件，则此函数将返回 cudaErrorInvalidValue。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInitializationError；cudaErrorPriorLaunchFailure；cudaErrorInvalidResourceHandle

(3) cudaEventQueryo

函数功能：查询是否已经记录了事件。

调用形式：cudaErrortcudaEventQuery(cudaEventtevent)

函数说明：如果确实已经记录了事件，则返回 cudaSuccess；如果尚未记录，则返回 cudaErrorNotReady。如果尚未对此事件调用 cudaEventRecord()，则该函数将返回 cudaErrorInvalidValue。

返回值：cudaSuccess；cudaErrorNotReady；cudaErrorInitializationError；cudaErrorPriorLaunchFailure；cudaErrorInvalidValue；cudaErrorInvalidResourceHandle

(4) cudaEventSynchronize

函数功能：等待事件被记录。

调用形式：cudaError_t cudaEventSynchronize(cudaEvent_t event)

函数说明：在实际记录事件之前，一直阻塞操作。如果尚未为此事件调用 cudaEventRecord()，则该函数返回 cudaErrorInvalidValue。

返回值：cudaSuccess；cudaErrorInitializationError；cudaErrorPriorLaunchFailure；cudaErrorInvalid；Value；cudaErrorInvalidResourceHandle

(5) cudaEventDestroy

函数功能：销毁事件对象。

调用形式：cudaError_t cudaEventDestroy(cudaEvent_t event)

返回值：cudaSuccess；cudaErrorInitializationError；cudaErrorPriorLaunchFailure；cudaErrorInvalidValue

(6) cudaEventElapsedTime

函数功能：计算两次事件之间相差的时间。

调用形式：cudaError_t cudaEventElapsedTime(float * time，cudaEvent_t start，cudaEvent_t end)。

函数说明：计算两次事件之间相差的时间(以毫秒为单位，精度为 0.5μs)。如果尚未记录其中任何一个事件，则此函数将返回 cudaErrorInvalidValue。如果记录其中任何一个事

件使用了非零流，则结果不确定。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInitializationError；cudaErrorPriorLaunchFailure；cudaErrorInvalidValue；cudaErrorInvalidResourceHandle

5．存储器管理

存储器管理主要用于管理主机内存和 GPU 显存的分配、初始化和释放，以及这两种不同存储器数据之间的复制。GPU 上的显存包括线性存储器和 CUDA 数组，针对不同类型的显存有不同的使用方法。与存储器管理相关的函数有以下几个：

（1）cudaMalloc

函数功能：在 GPU 上分配存储器。

调用形式：cudaError_t cudaMalloc(void ** devPtr，size_t count)

函数说明：向设备分配 count 字节的线性存储器，并以 * devPtr 的形式返回指向所分配存储器的指针。可针对任何类型的变量合理调整所分配的存储器，存储器不会被清除。如果出现错误，cudaMalloc()将返回 cudaErrorMemoryAllocation。

返回值：cudaSuccess；cudaErrorMemoryAllocation

（2）cudaMallocPitch

函数功能：向 GPU 分配存储器。

调用形式：cudaError_t cudaMallocPitch(void ** devPtr，size_t * pitch，size_t widthInBytes，size_t height)

函数说明：向设备分配至少 widthInBytes * height 字节的线性存储器，并以 * devPtr 的形式返回指向所分配存储器的指针。该函数可以填充所分配的存储器，以确保在地址从一行更新到另一行时，给定行的对应指针依然满足对齐要求。

cudaMallocPitch()以 * pitch 的形式返回间距，即所分配存储器的宽度，以字节为单位。间距是存储器分配时的一个独立参数，用于在二维数组内计算地址。如果给定一个 T 类型数组元素的行和列，则可按如下方法计算地址：

```
T * pElement = (T * )((char * )BaseAddress + Row * pitch) + Column;
```

对于二维数组的分配，建议使用 cudaMallocPitch()来执行间距分配。由于硬件中存在间距对齐的限制，如果应用程序将在设备存储器的不同区域之间执行二维存储器复制（无论线性存储器还是 CUDA 数组），这种方法将非常有用。

返回值：cudaSuccess；cudaErrorMemoryAllocation

（3）cudaFree

函数功能：释放 GPU 上的存储器。

调用形式：cudaError_t cudaFree(void * devPtr)

函数说明：释放 devPtr，dev 必须是之前调用 cudaMalloc()或 cudaMallocPitch()时返回所指向的存储器空间。如果未返回或者已经调用过 cudaFree(decPtr)，则返回一个错误。如果 devPtr 为 0，则不执行任何操作。如果出现错误，则 cudaFree()将返回 cudaErrorInvalidDevicePointer。

返回值：cudaSuccess；cudaErrorInvalidDevicePointer；cudaErrorInitializationError

（4）cudaMallocArray

函数功能：向 GPU 分配数组。

调用形式：cudaError_t cudaMallocArray（struct cudaArray ** array，const struct cudaChannelFormatDesc * desc，size_t width，size_t height ）

函数说明：根据 cudaChannelFormatDesc 的结构 desc 分配 CUDA 数组，以 * array 的形式返回新 CUDA 数组的句柄。

cudaChannelFormatDesc 定义如下：

struct cudaChannelFormatDesc{ int x,y,zw; enum cudaChannelFormatKind f; };

其中，cudaChannelFormatKind 是 cudaChannelFormatKindSigned、cudaChannelFormatKindUnsigned 或 cudaChannelFormatKindFloat 之一。

返回值：cudaSuccess；cudaErrorMemoryAllocation

（5）cudaFreeArray

函数功能：释放 GPU 上的数组。

调用形式：cudaError_t cudaFreeArray(struct cudaArray * array)

函数说明：释放 CUDA 数组 array。如果 array 为 0，则不执行任何操作。

返回值：cudaSuccess；cudaErrorInitializationError

（6）cudaMallocHost

函数功能：向主机分配分页锁定的存储器。

调用形式：cudaError_t cudaMallocHost(void ** hostPtr,size_t size)

函数说明：分配 size 字节大小的分页锁定且设备可访问的主存储器。驱动程序会跟踪由此函数分配的虚拟存储器范围，自动加速对 cudaMemcpy * ()等函数的调用。由于设备可直接访问存储器，所以与 malloc()等函数分配的可分页存储器相比，这种存储器在读取或写入时的带宽更高。使用 cudaMallocHost()分配过多存储器可能会导致系统性能降低，因为这将减少系统可用于分页的存储器数量。因而，尽量少使用此函数，一般只用于为主机和设备之间的数据交换分配存储器。

返回值：cudaSuccess；cudaErrorMemoryAllocation

（7）cudaFreeHost

函数功能：释放分页锁定的存储器。

调用形式：cudaError_t cudaFreeHost(void * hostPtr)

函数说明：释放 hostPtr 指向的存储器空间，之前的 cudaMallocHost()调用必须返回 hostPtr。

返回值：cudaSuccess；cudaErrorInitializationError

（8）cudaMemset

函数功能：初始化（设置）GPU 存储器的值。

调用形式：cudaErrortcudaMemset(void * devPtr,intvalue,sizetcount)

函数说明：使用固定字节值 value 来填充 devPtr 所指向存储器区域的前 count 个字节。

返回值：cudaSuccess；cudaErrorInvalid：Value；cudaErrorInvalidDevicePointer

（9）cudaMemset2D

函数功能：初始化 GPU 存储器的值。

调用形式：cudaErrortcudaMemset2D(void * devPtr, size_t dpitch, int value, size_t width, size_t height)

函数说明：将dstPtr指向的矩阵(共有height行，每行width字节)设置为指定值value。pitch是dstPtr指向的二维数组在存储器中的宽度，其中包括添加到各行末尾的填充符。在cudaMallocPitch()已经传回所使用的间距时，此函数的执行速度最快。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidDevicePointer

(10) cudaMemcpy

函数功能：在GPU和主机之间复制数据。

调用形式：cudaError_t cudaMemcpy(void * dst, const void * src, size_t count, enumcudaMemcpyKind kind)和cudaError_t cudaMemcpyAsync(void * dst, const void * src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream)

函数说明：从scr指向的存储器区域中，将count个字节复制到dst指向的存储器区域，其中kind是cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost或cudaMemcpyDeviceToDevice之一，用于指定复制的方向。存储器区域不可重叠。调用cudaMemcpy()时，如果dst和src指针与复制的方向不匹配，则将导致不确定的行为。

cudaMemcpyAsync()是异步的，可选择传入非零流参数，从而将其关联到一个流。它仅对分页锁定的主存储器有效，如果传入指向可分页存储器的指针，那么将返回一个错误。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidDevicePointer；cudaErrorInvalidMemcpyDirection

(11) cudaMemcpy2D

函数功能：在主机和设备之间复制数据。

调用形式：cudaError_t cudaMemcpy2D(void * dst, size_t dpitch, const void * src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)和(cudaMemcpy2DAsync(void * dst, size_t const, void * src, size_t spitch, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream)

函数说明：从src指向的存储器区域中将一个矩阵(共height行，每行width字节)复制到dst指向的存储器区域，其中kind是cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost或cudaMemcpyDeviceToDevice之一，用于指定复制的方向。dpitch和spitch是dst和src指向的二维数组在存储器中的宽度(以字节为单位)，其中包括添加到各行末尾的填充符。存储器区域不可重叠。调用cudaMemcpy2D()时，如果dst和src指针与复制方向不匹配，则将导致不确定的行为。如果dpitch或spitch超过允许的最大值，则cudaMemcpy2D()将返回一个错误。

cudaMemcpy2DAsync()是异步的，可选择传入非零流参数，从而将其关联到一个流。它仅对分页锁定的主存储器有效，如果传入指向可分页存储器的指针，那么将返回一个错误。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidpitchValue；cudaErrorInvalidDevicePointer；cudaErrorInvalidMemcpyDirection

(12) cudaMemcpyToArray

函数功能：在主机和设备间复制数据。

调用形式：cudaError_t cudaMemcpyToArray(struct cudaArray * dstArray,size_t dstX,size_t dstY,const void * src,size_t count,enum cudaMemcpyKind kind,cudaStream_t stream)

函数说明：从 src 指向的存储器区域内将 count 个字节复制到一个 CUDA 数组 dstArray,该数组的左上角从(dstX,dstY)开始,其中 kind 是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一,用于指定复制的方向。cudaMemcpyToArrayAsync()是异步的,可选择传入非零流参数,从而将其关联到一个流。它仅对分页锁定的主存储器有效,如果传入指向可分页存储器的指针,那么将返回一个错误。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidDevicePointer；cudaErrorInvalidMemcpyDirection

(13) cudaMemcpy2DTOArray

函数功能：在主机和设备间复制数据。

调用形式：cudaError_t cudaMemcpy2DToArray(struct cudaArray * dstArray,size_t dstX,size_t dstY,const void * src,size_t spitch,size_t width,size_t height,enum cudaMemcpyKind kind)；

cudaError_t cudaMemcpy2DToArrayAsync (struct cudaArray * dstArray,size_t dstX,size_t dstY,const void * src,size_t spitch,size_t width,size_t height,enum cudaMemcpyKind kind,cudaStream_t stream)。

函数说明：从 src 指向的存储器区域内将一个矩阵(共 height 行,每行 width 字节)复制到 CUDA 数组 dstArray,该数组的左上角从(dstX,dstY)开始,其中 kind 是 cudaMemcpyHostTOHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一,用于指定复制的方向。spitch 是 src 指向的二维数组在存储器中的宽度(以字节为单位),其中包括添加到各行末尾的填充符。如果 spitch 超过允许的最大值,则 cudaMemcpy2D()将返回一个错误。

cudaMemcpy2DToArrayAsync()是异步的,可选择传入非零流参数,从而将其关联到一个流。它仅对分页锁定的主存储器有效,如果传入指向可分页存储器的指针,那么将返回一个错误。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidDevicePointer；cudaErrorInvalidPitchValue；cudaErrorInvalidMemcpyDirection

(14) cudaMemcpyFromArray

函数功能：在主机和设备间复制数据。

调用形式：cudaError_t cudaMemcpyFromArray(void * dst,const struct cudaArray * srcArray,size_t srcX,size_t srcY,size_t count,enum cudaMemcpyKind kind)；

cudaError_t cudaMemcpyFromArrayAsync(void * dst,const struct cudaArray * srcArray,size_t srcX,size_t srcY,size_t count,enum cudaMemcpyKind kind,cudaStream_t stream)；

函数说明：从一个 CUDA 数组 dstArray(左上角以(srcX,srcY)开始)内将 count 个字节复制到 dst 指向的存储器区域。其中 kind 是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice、

cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一，用于指定复制的方向。

cudaMemcpyFromArrayAsync()是异步的，可选择传入非零流参数，从而将其关联到一个流。它仅对分页锁定的主存储器有效，如果传入指向可分页存储器的指针，那么将返回一个错误。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidDevicePointer；cudaErrorInvalidMemcpyDirection

(15) cudaMemcpy2DFromArray

函数功能：在主机和设备间复制数据。

调用形式：cudaError_t cudaMemcpy2DFromArray(void * dst, size_t dpitch, const struct cudaArray * srcArray, size_t srcX, size_t srcY, size_t width, size_t height, enum cudaMemcpyKind kind)；

cudaError_t cudaMemcpy2DFromArrayAsync(void * dst, size_t dpitch, const struct cudaArray * srcArray, size_t srcX, size_t srcY, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream)；

函数说明：从一个 CUDA 数组 dstArray(左上角以(srcX, srcY)开始)内将一个矩阵(共 height 行，每行 width 字节)复制到 dst 指向的存储器区域，其中 kind 是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一，用于指定复制的方向。dpitch 是 dst 指向的二维数组在存储器中的宽度(以字节为单位)，其中包括添加到各行末尾的填充符。如果 dpitch 超过允许的最大值，则 cudaMemcpy2D()将返回一个错误。

cudaError_t cudaMemcpy2DFromArrayAsync 是异步的，可选择传入非零流参数，从而将其关联一个流。它仅对分页锁定的主存储器有效，如果传入指向可分页存储器的指针，那么将返回一个错误。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidDevicePointer；cudaErrorInvalidPitchValue；cudaErrorInvalidMemcpyDirection

(16) cudaMemcpyArrayToArray

函数功能：在主机和设备间复制数据。

调用形式：cudaError_t cudaMemcpyArrayTOArray (struct cudaArray * dstArray, size_t dstX, size_t dstY, const struct cudaArray * srcArray, size_t srcX, size_t srcY, size_t count, enum cudaMemcpyKind kind)

函数说明：从一个 CUDA 数组 dstArray(左上角以(srcX, srcY)开始)内将 count 字节复制到另一个 CUDA 数组 dstArray(左上角以(dstX, dstY)开始)，其中 kind 是 cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyHostToHost、cudaMemcpyDeviceToDevice 之一，用于指定复制的方向。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidMemcpyDirection

(17) cudaMemcpy2DArrayTOArray

函数功能：在主机和设备间复制数据。

调用形式：cudaError_t cudaMemcpy2DArrayToArray (struct cudaArray * dstArray, size_t dstX, size_t dstY, const struct cudaArray * srcArray, size_t srcX, size_t srcY, size_t

width,size_t height,enum cudaMemcpyKind kind)

函数说明：从一个 CUDA 数组 dstArray(左上角以(srcX,srcY)开始)内将一个矩阵(供 height 行,每行 width 字节)复制到另一个 CUDA 数组 dstArray(左上角以(dstX,dstY)开始),其中 kind 是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一,用于指定复制的方向。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidMemcpyDirection

(18) cudaMemcpyToSymbol

函数功能：将主存储器的数据复制到 GPU。

调用形式：template < class T >

cudaError_t cudaMemcpyToSymbol(const T&symbol,const void * src,size_t count,size_t offset,enum cudaMemcpyKind kind)

函数说明：从 src 指向的存储器区域内将 count 个字节复制到从 symbol 偏移 offset 字节,所指向的存储器区域。存储器区域不可重叠。symbol 可以是位于全局存储器或不变存储器空间内的变量,也可以是一个指定全局存储器或不变存储器空间变量的字符串。kind 可以是 cudaMemcpyHostToDevice 或 cudaMemcpyDeviceToDevice。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidSymbol；cudaErrorInvalidDevicePointer；cudaErrorInvalidMemcpyDirection

(19) cudaMemcpyFromSymb

函数功能：将 GPU 的数据复制到主存储器。

调用形式：template < class T >

cudaError_t cudaMemcpyFromSymbol(void * dst,const T & symbol,size_t count,size_t offset,enum cudaMemcpyKind kind)

函数说明：从 offset 字节所指向的存储器区域内(从 symbol 符号开始)将 count 个字节复制到 dst 指向的存储器区域。存储器区域不可重叠。symbol 可以是位于全局存储器或不变存储器空间内的变量,也可以是一个指定全局存储器或不变存储器空间变量的字符串。kind 可以是 cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidSymbol；cudaErrorInvalidDevicePointer；cudaErrorInvalidMemcpyDirection

(20) cudaGetSymbolAddress

函数功能：查找与 CUDA 符号关联的地址。

调用形式：template < class T >

cudaError_t cudaGetSymbolAddress(void ** devPtr,const T & symbol)

函数说明：以 * devPtr 的形式返回符号 symbol 在设备上的地址。symbol 可以是位于全局存储器或不变存储器空间内的变量,也可以是一个指定全局存储器或不变存储器空间变量的字符串。如果无法找到 symbol,或未在全局存储器空间内声明 symbol,* devPtr 将保持不变,并返回一个错误。如果出现错误,则 cudaGetSymbolAddress()将返回 cudaErrorInvalidSymbol。

返回值：cudaSuccess；cudaErrorInvalidSymbol；cudaErrorAddressOfConstant

(21) cudaGetSymbolSize

函数功能：查找与 CUDA 符号关联的对象的大小。

调用形式：template＜class T＞

cudaError_t cudaGetSymbolSize (size_t * size,const T & symbol)

函数说明：以 * size 的形式返回符号 symbol 的大小。symbol 可以是位于全局存储器或不变存储器空间内的变量，也可以是一个指定全局存储器或不变存储器空间变量的字符串。如果无法找到 symbol，或未在全局或不变存储器空间内声明 symbol，* size 将保持不变，并返回一个错误。如果出现错误，则 cudaGetSymbolSize()将返回 cudaErrorInvalidSymbol。

返回值：cudaSuccess；cudaErrorInvalidSymbol

(22) cudaMalloc3D

函数功能：向 GPU 分配逻辑一维、二维或三维存储器对象。

调用形式：

```
struct cudaPitchedPtr {
          void * ptr;
          size_t pitch;
          size_t xsize;
          size_t ysize; };
struct cudaExtent{
          size_t width;
          size_t height;
          size_t depth; };
cudaError_t cudaMalloc3D ( struct cudaPitchedPtr * pitchDevPtr,struct cudaExtent extent)
```

函数说明：向设备分配至少 width * height * depth 个字节的线性存储器，并返回 pitchedDevPtr，其中 ptr 是指向已分配存储器的指针。该函数可填充所分配的存储器，确保满足硬件对齐要求。pitchedDevPtr 的 pitch 字段中返回的间距是所分配存储器的宽度，以字节为单位。返回的 cudaPitchedPtr 包含额外的字段 xsize 和 ysize，是所分配存储器的逻辑宽度和高度。对于二维或三维对象的分配，建议使用 cudaMalloc3D()或 cudaMallocPitch()来执行分配。由于硬件中存在对齐限制，如果应用程序将执行涉及二维或三维对象的存储器复制，那么这种方法将非常有用。

返回值：cudaSuccess；cudaErrorMemoryAllocation

(23) cudaMalloc3DArray

函数功能：向 GPU 分配一个数组。

调用形式：struct cudaExtent{size_t width；size_t height；size_t depth；}；

cudaError_t cudaMalloc3DArray (struct cudaAray ** arrayPtr，const struct cudaChannelFomatDesc * desc，struct cudaExtent extent)

函数说明：根据 cudaChannelFormatDesc 结构 desc 分配一个 CUDA 数组，并以 * arrayPtr 的形式返回新 CUDA 数组的句柄。

cudaChannelFormatDesc 定义如下：

```
struct cudaChannelFormatDesc { int x,y,z,w;
       enum cudaChannelFormatKind f; };
```

其中，cudaChannelFormatKind 是 cudaChannelFormatKindSigned、cudaChannelFormatKindUnsigned 或 cudaChannelFormatKindFloat 之一。

cudaMalloc3Darray 能够分配一维、二维或三维数组。

① 如果 height 和 depth 范围都是 0，则分配一维数组。对于一维数组，有效范围是{(1，8192)，0，0}。

② 如果仅有 depth 范围是 0，则分配二维数组。对于二维数组，有效范围是{(1，65536)，(1，32768)，0}。

③ 如果三项范围全部为非 0，则分配三维数组。对于三维数组，有效范围是{(1，2048)，(1，2048)，(1，2048)}。

注意，区分范围限制有助于利用更高维度的退化数组。例如，退化二维数组比一维数组支持的线性存储空间更多。

返回值：cudaSuccess；cudaErrorMemoryAllocation

(24) cudaMemset3D

函数功能：初始化 GPU 存储器的值。

调用形式：

```
struct cudaPitchedPtr {
        void * ptr;
        size_t pitch;
        size_t xsize;
        size_t ysize; };
struct cudaExtent {
        size_t width;
        size_t height;
        size_t depth; };
cudaError_t cudaMemset3D ( struct cudaPitchedPtr dstPitchPtr, int value, struct cudaExtent extent )
```

函数说明：初始化三维数组的各元素，将其设置为特定值 value。dstPitchPtr 定义了需要初始化的对象。dstPitchPtr 的 pitch 字段是 dstPitchPtr 指向的三维数组在存储器中的宽度(以字节为单位)，其中包括添加到各行末尾的填充符。xsize 字段指定各行的逻辑宽度(以字节为单位)；ysize 字段指定二维片段的高度，以行为单位。

被初始化的区域的范围指定如下：宽度为 width 字节；高度为 height 行；深度为 depth。

如果区域的 width 大于等于 dstPitchPtr 的 xsize，则其执行速度将远远超过宽度小于 xsize 的区域。其次，如果区域的 height 等于 dstPitchPtr 的 ysize，则其执行速度将超过高度小于 ysize 的区域。

在使用 cudaMalloc3D()分配了 dstPitchPtr 的情况下，此函数的执行速度最快。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidDevicePointer

(25) cudaMemcpy3D

函数功能：在三维对象间复制数据。

调用形式：

```
struct cudaExtent {
        size_t width,height,depth; };
struct cudaPos{
        size_t s,y,z; };
```

```
struct cudaMencpy3DParms {
        struct cudaArray * srcArray;
        struct cudaPos srcPos;
        struct cudaPitchedPtr srcPtr;
        struct cudaArray * dstArray;
        struct cudaPos dstPos;
        struct cudaPitchedPtr dstPtr;
        struct cudaExtent extent;
        enum cudaMemcpyKind kind; };
cudaError_t cudaMemcpy3D(const struct cudaMemcpy3DParms * p)
cudaError_t cudaMemcpy3DAsync(const struct cudaMemcpy3DParms * p, cudaStream_t stream)
```

函数说明：cudaMemcpy3D()在两个三维对象间复制数据。源对象和目标对象均可以是主存储器、设备存储器或 CUDA 数组。所执行的复制操作的源、目标、范围和类型由 cudaMemcpy3Dparms 结构体指定，在使用之前应将其初始化为 0：

```
cudaMemcpy3DParms myParms = { 0 };
```

传递给 cudaMemcpy3D()的结构体必须指定 srcArray 或 srcPtr 以及 dstArray 或 dstPtr。传递多个非零源或目标将导致 cudaMemcpy3D()返回一个错误。

srcPos 和 dstPos 字段是源和目标对象的可选偏移，是在各对象元素的单位中定义的。主机或设备指针的元素应为无符号字符。对于 CUDA 数组，指针的所有维度都必须在[0,2048)。

extent 字段定义元素中传输区域的维度。如果一个数组参与复制，则范围将根据该数组元素进行定义。如果没有任何 CUDA 数组参与复制，则范围将以无符号字符元素定义。

kind 字段定义复制的方向。它必须是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一。

如果源和目标均为数组，且元素大小不同，则 cudaMemcpy3D()将返回一个错误。

源和目标对象不可重叠。如果指定的源和目标对象重叠，则将导致不确定的行为。

如果 srcPtr 或 dstPtr 的间距超过允许的最大值，cudaMemcpy3D()将返回一个错误。使用 cudaMalloc3D()分配的 cudaPitchedPtr 的间距总是有效的。

cudaMemcpy3DAsync()是一种异步复制操作，可选择传入非零流参数，从而将其关联到一个流。如果源或目标是主对象，则必须将其分配到 cudaMallocHost()返回的分页锁定存储器中。如果传入了一个未使用 cudaMallocHost()进行分配的存储器指针，那么它将返回一个错误。

返回值：cudaSuccess

6. 纹理引用管理

纹理存储器属于只读存储器，但其拥有缓存机制。它能通过缓存并利用数据的局部性来提高效率。纹理引用管理包括了纹理参考的创建、绑定存储器到纹理、解除纹理绑定等功能函数。与纹理存储器管理相关的函数如下：

(1) cudaCreateChannelDesc

函数功能：通道描述符。

调用形式：template < class T >

struct cudaChannelFormatDesc cudaCreateChannelDesc < T >()

函数说明：返回通道描述符，格式为 f，各组件的位数为 x、y、z 和 w。cudaChannelFormatDesc 定义如下：

```
struct cudaChannelFormatDesc { int x,y,z,w;enum cudaChannelFormatKind f; };
```

其中，cudaChannelFormatKind 是 cudaChannelFormatKindSigned、cudaChannelFormatKindUnsigned 或 cudaChannelFormatKindFloat 之一。

返回值：cudaSuccess

（2） cudaBindTexture

函数功能：将存储器绑定到纹理。

调用形式：template < class T，int dim，enum cudaTextureReadMode readMode >

static __inline__ __host__cudaError_t cudaBindTexture (sizet * offset，const struct texture < T，dim，readMode > & texRef，const void * devPtr，const struct cudaChannelFormatDesc&desc，size_t size = UINT_MAX)

函数说明：将 devPtr 指向的存储器区域中的 size 个字节绑定到纹理引用 texRef。desc 描述从纹理中获取值时如何解释存储器。offset 参数是可选的偏移字节，这与低级 cudaBindTexture()函数相同。所有以前绑定到 texRef 的存储器都将解除绑定。

template < class T，int dim，enum cudaTextureReadMode readMode >

static __inline__ __ host__ cudaError_t cudaBindTexture(size_t * offset，const struct texture < T，dim，readMode > & texRef，const void * devPtr，size_t size = UINT_MAX)

将 devPtr 指向的存储器区域中的 size 个字节绑定到纹理引用 texRef。通道描述符继承自纹理引用类型。offset 参数是可选的字符偏移。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidDevicePointer；cudaErrorInvalidTextureo

（3） cudaBindTextureToArray

函数功能：将 CUDA 数组绑定到纹理。

调用形式：template < class T，int dim，enum cudaTextureReadMode readMode >

static __inline__ __host__ cudaError_t cudaBindTextureToArray (const struct texture < T，dim，readMode > & texRef，const struct cudaArray * cuArray)

函数说明：将 CUDA 数组 array 绑定到纹理引用 texRef。desc 描述从纹理中获取值时如何解释存储器。所有以前绑定到 texRef 的 CUDA 数组都将解除绑定。

template < class T，int dim，enum cudaTextureReadMode readMode >

static __inline__ __host__ cudaError_t cudaBindTextureToArray(const struct texture < T，dim，readMode > & texRef，const struct cudaArray * cuArray)

将 CUDA 数组 array 绑定到纹理引用 texRef。通道描述符继承自 CUDA 数组。所有以前绑定到 texRef 的 CUDA 数组都将解除绑定。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidDevicePointer；cudaErrorInvalidTexture

（4） cudaUnbindTexture

函数功能：解除纹理绑定。

调用形式：template < class T,int dim,enum cudaTextureReadMode readMode >

static__inline__ __host__ cudaError_t cudaUnbindTexture (const struct texture < T,dim,readMode > & texRef)

函数说明：将绑定到纹理引用 texRef 的纹理解除绑定。

返回值：cudaSuccess

7. 执行控制管理

执行控制管理包括配置设备启动、启动设备等功能，其相关的函数如下：

(1) cudaConfigureCall

函数功能：配置设备启动。

调用形式：cudaErorr_t cudaConfigureCall (dim3 gridDim, dim3 blockDim, size_t sharedMem=0,int tokens = 0)

函数说明：为要执行的设备调用指定网格和块大小，类似于执行配置语法。cudaConfigureCall()基于堆栈。每次调用都会将数据放入一个执行堆栈的顶端。此数据包含线程网格和线程块的大小和针对调用的所有参数。

返回值：cudaSuccess；cudaErrorInvalidConfiguration

(2) cudaLaunch

函数功能：启动设备函数。

调用形式：template < class T > cudaError_t cudaLaunch (T entry)

函数说明：在设备上启动函数 entry。entry 可以是一个在设备上执行的函数，也可以是指定在设备上执行的函数的字符串。entry 必须声明为全局函数。在 cudaLaunch ()之前必须存在 cudaConfigureCall()调用，因为它将从执行堆栈中弹出 cudaConfigureCall()放入的数据。

返回值：cudaSuccess；cudaErrorInvalidDeviceFunction；cudaErrorInvalidConfiguration

(3) cudaSetupArgument

函数功能：配置设备启动。

调用形式：cudaError_t cudaSetupArgument (void * arg,size_t count,size_t offset) template < class T > cudaError_t cudaSetupArgument (T arg,size_t offset)

函数说明：将 arg 指向的参数中的 count 个字节放到距离区域传递参数 offset 个字节处，其中区域从 offset = 0 开始。参数存储在执行堆栈顶端。cudaSetupArgument()之前必须存在 cudaConfigureCall()调用。

返回值：cudaSuccess

8. 图形学互操作性

一些 OpenGL 和 Direct3D 的资源可被映射到 CUDA 地址空间，CUDA 可以读 OpenGL 或 Direct3D 写的数据，CUDA 也可以写数据供 OpenGL 或 Direct3D 使用。

资源必须先在 CUDA 中注册，才能在 CUDA 中被映射。映射的函数返回一个指向 cudaGraphicsResource 类型结构体的 CUDA 图形资源。资源注册是潜在高消耗的，因此通常每个资源只注册一次。可以使用 cudaGraphicsUnregisterResource()解注册 CUDA 图形资源。

一旦资源被注册到 CUDA，就可以按需要被任意次地映射和解映射，映射和解映射使

用 cudaGraphicsMapResources ( ) 和 cudaGraphicsUnmapResources ( )。可以使用 cudaGraphicsResourceSetMapFlags()来指定资源用处(只读,只写),CUDA 驱动可以据此优化资源管理。

通过 cudaGraphicsResourceGetMappedPointer()可以获得缓冲区返回的设备地址空间,通过 cudaGraphicsSubResourceGetMappedArray()可以获得 CUDA 数组返回的设备地址空间,内核通过读写这些空间操作被映射资源。

图形学互操作性相关的函数如下:

(1) cudaGraphicsMapResources

函数功能:映射图形资源,以便 CUDA 访问。

调用形式:cudaError_t cudaGraphicsMapResources (int count,cudaGraphicsResource_t * resources,cudaStream_t stream=0)

函数说明:映射 resources 中的 count 图形资源,以便 CUDA 访问。

解映射之前,可在 CUDA 内核中访问 resources 中的资源。在 CUDA 映射了资源后,已被注册的 resources 的图形 API 不应访问任何资源。如果应用程序允许其访问,将导致不确定的结果。

此函数提供了同步保证,确保在它开始执行 CUDA 内核之前的任何图形调用已完成。

如果 resources 包含重复项,则将返回 cudaErrorInvalidResourceHandle。如果已有 resources 映射为 CUDA 访问,则返回 cudaErrorUnknown。

返回值:cudaSuccess;cudaErrorlnvalidResourceHandle;cudaErrorUnknown

(2) cudaGraphicsResourceGetMappedMipmappedArray

函数功能:获取已映射图形资源对应的 mipmap 数组。

调用形式:cudaError_t cudaGraphicsResourceGetMappedMipmappedArray (cudaMipmappedArray_t * mipmappedArray,cudaGraphicsResource_t resource)

函数说明:以 * mipmappedArray 的形式返回已映射图形资源对应的 mipmap 数组。每次映射 resource 时,* mipmappedArray 中设置的值都会发生变化。

如果 resource 不是一个纹理,则无法通过阵列存取,函数返回 cudaErrorUnknown。如果 resource 没有被映射,则函数返回 cudaErrorUnknown。

返回值:cudaSuccess;cudaErrorInvalidValue;cudaErrorInvalidResourceHandle;cudaErrorUnknown

(3) cudaGraphicsResourceGetMappedPointer

函数功能:获取已映射图形资源对应的设备指针。

调用形式:cudaError_t cudaGraphicsResourceGetMappedPointer(void ** devPtr,size_t * size,cudaGraphicsResource_t resource)

函数说明:以 * devPtr 的形式返回已映射图形资源 resource 对应的指针。以 * size 的形式返回从 * devPtr 指针处可访问的内存大小的字节数。每次映射 resource 时,* devPtr 中设置的值都会发生变化。

如果 resource 不是一个缓冲区,则无法通过指针被访问,函数返回 cudaErrorUnknown。如果 resource 没有被映射,则函数返回 cudaErrorUnknown。

返回值:cudaSuccess;cudaErrorInvalidValue;cudaErrorInvalidResourceHandle;

cudaErrorUnknown

（4）cudaGraphicsResourceSetMapFlags

函数功能：设置映射图形资源的使用标志。

调用形式：cudaError_t cudaGraphicsResourceSetMapFlags (cudaGraphicsResource_t resource，unsigned int flags）

函数说明：设置映射图形资源 resource 的标志位。

更改 flags 将在下一次 resource 映射时生效。flags 参数可以是下列任何一项。

① cudaGraphicsMapFlagsNone：未指定 resource 将被如何使用。因此，假设 CUDA 可读取或写入 resource。

② cudaGraphicsMapFlagsReadOnly：指定 CUDA 不会写入 resource。

③ cudaGraphicsMapFlagsWriteDiscard：指定不会读取 resource，且将改写 resource 的全部内容，因此预先存储在 resource 中的数据都不被保留。

如果 resource 是目前映射以便 CUDA 访问，则函数返回 cudaErrorUnknown。如果 flags 不是上述值之一，则函数返回 cudaErrorInvalidValue。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidResourceHandle；cudaErrorUnknown

（5）cudaGraphicsSubResourceGetMappedArray

函数功能：获得已映射图形资源的子资源对应的数组。

调用形式：cudaError_t cudaGraphicsSubResourceGetMappedArray(cudaArray_t * array，cudaGraphicsResource_t resource，unsigned int arrayIndex，unsigned int mipLevel）

函数说明：以 * array 的形式返回对应于数组索引 arrayIndex 和 mipmap 等级 mipLevel 的已映射图形资源 resource 的子资源的数组。每次映射 resource 时，array 中设置的值都会发生变化。

如果 resource 不是纹理，则无法通过数组访问，函数将返回 cudaErrorUnknown。如果 arrayIndex 不是 resource 的一个有效的数组索引，函数将返回 cudaErrorInvalidValue。如果 mipLevel 不是 resource 的一个有效的 mipmap 等级，那么函数将返回 cudaErrorInvalidValue。如果 resource 没有被映射，那么函数将返回 cudaErrorUnknown。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorInvalidResourceHandle；cudaErrorUnknown

（6）cudaGraphicsUnmapResources

调用形式：cudaError_t cudaGraphicsUnmapResources(int count，cudaGraphicsResource_t * resources，cudaStream_t stream=0）

函数功能：解映射图形资源。

返回值：cudaSuccess；cudaErrorInvalidResourceHandle；cudaErrorUnknown

函数说明：解映射 resources 中的 count 图形资源。

一旦解映射，resources 中的图形资源不能被 CUDA 访问，直到它们再次被映射。此函数提供了同步保证，确保在 cudaGraphicsUnmapResources 开始发出图形调用之前，cudaGraphicsUnmapResources 之前发出的任何 CUDA 内核都将完成。

如果 resources 包含重复项，则将返回 cudaErrorInvalidResourceHandle。如果 resources 中

目前有未映射为 CUDA 访问的资源，则返回 cudaErrorUnknown。

(7) cudaGraphicsUnregisterResource

函数功能：注销图形资源。

调用形式：cudaError_t cudaGraphicsUnregisterResource (cudaGraphicsResource_t resource)

函数说明：注销图形资源 resource，除非再次注册，否则 CUDA 将无法再访问此资源。如果 resource 是无效的，则函数返回 cudaErrorInvalidResourceHandle。

返回值：cudaSuccess；cudaErrorInvalidResourceHandle；cudaErrorUnknown

9. OpenGL 互操作性

CUPA 和 OpenGL 的互操作要求，在其他任何运行时函数调用前使用 cudaGLSetGLDevice()指定 CUDA 设备。注意 cudaSetDevice()和 cudaGLSetDevice()是相互排斥的。

可以被映射到 CUDA 地址空间的 OpenGL 资源有 OpenGL 缓冲区、纹理和渲染缓存对象。

使用 cudaGraphicsGLRegisterBuffer()注册缓冲对象。在 CUDA 中，缓冲对象表现为设备指针，因此可以在内核中读写或通过 cudaMemcpy()调用。

纹理或渲染缓存对象使用 cudaGraphicsGLRegiSterlmage()注册。在 CUDA 中，它们表现为 CUDA 数组，可绑定到纹理参考，可被内核读写或通过 cudaMemcpy2D()调用。cudaGraphicsGLRegiSterlmage()使用内置的 float 类型(如 GL_RGBA_FLOAT32)和非归一化整数(如 GL_RGBA8UI)支持所有纹理格式。

与 OpenGL 互操作性相关的函数如下：

(1) cudaGLGetDevices

函数功能：获取与当前的 OpenGL 上下文关联的 CUDA 设备。

调用形式：cudaError_t cudaGLGetDevices(unsigned int * pCudaDeviceCount，int * pCudaDevices，unsigned int cudaDeviceCount，cudaGLDeviceList deviceList)

函数说明：返回值 * pCudaDeviceCount 对应于当前的 OpenGL 上下文 CUDA 兼容的设备的数量。返回值 * pCudaDevices 的 cudaDeviceCount 对应于当前 OpenGL 上下文的最多 CUDA 兼容设备数量。如果任何正在使用的当前 OpenGL 上下文的图形处理器不支持 CUDA 功能，则该函数调用将返回 cudaErrorNoDevice。

返回值：cudaSuccess；cudaErrorNoDevice；cudaErrorUnknown

(2) cudaGraphicsGLRegisterBuffer

函数功能：注册一个 OpenGL 缓冲区对象。

调用形式：cudaError_t cudaGraphicsGLRegisterBuffer(cudaGraphicsResource ** resource，GLuint buffer，unsigned int flags)

函数说明：注册由 buffer 指定的缓冲区对象，以便 CUDA 访问。注册对象的句柄返回为 resource。注册标志 flags 指定使用目的，如下所示：

① cudaGraphicsRegisterFlagsNone——未指定该资源将被如何使用。因此，假设此资源将由 CUDA 读取和写入。

② cudaGraphicsRegisterFlagsReadOnly——指定 CUDA 不会写入此资源。

③ cudaGraphicsRegisterFlagsWriteDiscard——指定 CUDA 不会读取此资源，且将改

写该资源的全部内容，所以现存于该资源中的数据将不被保留。

返回值：cudaSuccess；cudaErrorInvalidDevice；cudaErrorInvalidValue；ResourceHandle；cudaErrorUnknown

（3）cudaGraphicsGLRegisterImage

函数功能：注册一个 OpenGL 纹理或渲染缓冲区对象。

调用形式：cudaError_t cudaGraphicsGLRegisterImage（cudaGraphicsResource ** resource，GLuint image，GLenum target，unsigned int flags）

函数说明：注册由 image 所指定的纹理或渲染缓冲区对象，以便 CUDA 访问。注册对象的句柄返回为 resource。target 必须匹配对象的类型，并且必须是 GL_TEXTURE_2D、GL_TEXTURE_RECTANGLE、GL_TEXTURE_CUBE_MAP、GL_TEXTURE_3D、GL_TEXTURE_2D_ARRAY 或 GL_RENDERBUFFER 其中之一。

注册标志 flags 指定使用目的，如下所示：

① cudaGraphicsRegisterFlagsNone——未指定该资源将被如何使用。因此，假设此资源将由 CUDA 读取和写入。

② cudaGraphicsRegisterFlagsReadOnly——指定 CUDA 不会写入此资源。

③ cudaGraphicsRegisterFlagsWriteDiscard——指定 CUDA 不会读取此资源，且将改写该资源的全部内容，所以现存于资源中的数据将不被保留。

④ cudaGraphicsRegisterFlagsSurfaceLoadStore——指定 CUDA 将绑定这个资源到一个表面的参考。

⑤ cudaGraphicsRegisterFlagsTextureGather——指定 CUDA 将在该资源上执行纹理聚集操作。

简洁起见，以缩写形式表示所支持的图像格式。例如，{GL_R，GL_RG} X{8，16}将扩展为以下 4 种格式{GL_R8，GL_R16，GL_RG8，GL_RG16}：

① GL_RED，GL_RG，GL_RGBA，GL_LUMINANCE，GL_ALPHA，GL_LUMINANCE_ALPHA，GL_INTENSITY。

② {GL_R，GL_RG，GL_RGBA} X{8，16，16F，32F，8UI，32UI，8I，16I，32I}。

③ {GL_LUMINANCE，GL_ALPHA，GL_LUMINANCE_ALPHA，GL_INTENSITY} X {8，16，16F_ARB，32F_ARB，8UI_EXT，16UI_EXT，32UI_EXT，8I_EXT，16I_EXT，32I_EXT）。

返回值：cudaSuccess；cudaErrorInvalidDevice；cudaErrorInvalidValue；cudaErrorInvalidResourceHandle；cudaErrorUnknown

（4）cudaWGLGetDevice

函数功能：获取与 hGpu 相关的 CUDA 设备。

调用形式：cudaError_t cudaWGLGetDevice(int * device，HGPUNV hGpu)

函数说明：返回与 hGpu 相关的 CUDA 设备。

返回值：cudaSuccess

10. Direct3D 互操作性

Direct3D 互操作性支持 Direct3D9、Direct3D10 和 Direct3D11。

CUDA 一次只能和一个 Direct3D 设备互操作，且 CUDA 和 Direct3D 设备必须在同一个

GPU 上创建，同时，Direct3D 设备必须使用 D3DCREATE_HARDWARE_VERTEXPROCESSING 标签创建。

CUDA 和 Direct3D 的互操作要求：在任何其他的运行时函数调用前，使用 cudaD3D9SetDirect3DDevice()、cudaD3D10SetDirect3DDevice()和 cudaD3D11SetDirect3DDevice()指定 Direct3D 设备。可用 cudaD3D9GetDevice()、cudaD3D10GetDevice()和 cudaD3D11GetDevice()检索并关联到一些适配器的 CUDA 设备。

可以被映射到 CUDA 地址空间的 Direct3D 资源有 Direct3D 缓冲区、纹理和表面。可以使用 cudaGraphicsD3D9RegisterResource()、cudaGraphicsD3D10RegisterResource()和 cudaGraphicsD3D11RegisterResource()注册这些资源。

（1）cudaD3D10GetDevice

函数功能：获取适配器的设备号。

调用形式：cudaError_t cudaD3D10GetDevice(int * device, IDXGIAdapter * pAdapter)

函数说明：返回对应于由 IDXGIFactory::EnumAdapters 获得的适配器 pAdapter 的 CUDA 兼容设备 * device，仅当适配器 pAdapter 对应的设备是 CUDA 兼容的，该调用才会成功。

返回值：cudaSuccess；cudaErrorInvalidValue；cudaErrorUnknown

（2）cudaD3D10GetDevices

函数功能：获取对应于 Direct3D10 设备的 CUDA 设备。

调用形式：cudaError_t cudaD3D10GetDevices(unsigned int * pCudaDeviceCount, int * pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device * pD3D10Device, cudaD3D10DeviceList deviceList)

函数说明：返回 * pCudaDeviceCount 表示对应于 Direct3D 10 的设备 pD3D10Device 的 CUDA 兼容的设备数量。返回 * pCudaDevices 表示最多 cudaDeviceCount 个对应于 Direct3D 10 设备 pD3D10Device 的 CUDA 兼容设备。

如果任何被用来渲染 pDevice 的 GPU 不支持 CUDA，那么该调用将返回 cudaErrorNoDevice。

返回值：cudaSuccess；cudaErrorNoDevice；cudaErrorUnknown

（3）cudaGraphicsD3D10RegisterResource

函数功能：注册一个 Direct3D10 资源，以便 CUDA 访问。

调用形式：cudaError_t cudaGraphicsD3D10RegisterResource (cudaGraphicsResoure ** resource, ID3D10Resource * pD3DResource, unsigned int flags)

函数说明：注册 Direct3D 10 资源 pD3DResource 以便 CUDA 访问。

如果此调用成功，则应用程序能够在资源注销之前映射和解映射此资源，直到它通过调用函数 cudaGraphicsUnregisterResource()进行注销。

此调用开销较高，不应在交互式应用程序的每一帧中进行调用。

pD3DResource 的类型必须为以下之一：

① ID3D10Buffer——可以通过一个设备指针访问。

② ID3D10Texture1D——纹理的个体子资源可以通过数组访问。

③ ID3D10Texture2D——纹理的个体子资源可以通过数组访问。

④ ID3D10Texture3D——纹理的个体子资源可以通过数组访问。

Flags 参数可用于注册时指定其他参数，此参数的有效值如下：

① cudaGraphicsRegisterFlagsNone——未指定该资源将被如何使用。

② cudaGraphicsRegisterFlagsSurfaceLoadStore——指定 CUDA 将绑定该资源到一个表面的参考。

③ cudaGraphicsRegisterFlagsTextureGather——指定 CUDA 将在该资源上执行纹理聚集操作。

并非所有上述类型的 Direct3D 资源均可用于与 CUDA 进行互操作。下面列举了部分限制：

① 主呈现目标未向 CUDA 注册。

② 分配为共享的资源未向 CUDA 注册。

③ 纹理格式不是 1、2 或 4 个通道的 8 位、16 位或 32 位整数或浮点数的数据不能共享。

④ 深度或模板格式表面不能共享。

支持 DXGI 格式的完整列表如下，紧凑符号 A_{B,C,D}表示 A_B、A_C 和 A_D：

① DXGI_FORMAT_A8_UNORM。

② DXGI_FORMAT_B8G8R8A8_UNORM。

③ DXGI_FORMAT_R16_FLOAT。

④ DXGI_FORMAT_R16G16B16A16_{ FLOAT,SINT,SNORM,UINT,UNORM}。

⑤ DXGI_FORMAT_R16G16_{FLOAT,SINT,SNORM,UINT,UNORM }。

⑥ DXGI_FORMAT_R8_{ SINT,SNORM,UINT,UNORM }。

⑦ DXGI_FORMAT_R16_{ SINT,SNORM,UINT,UNORM }。

⑧ DXG1_FORMAT_R32_FLOAT。

⑨ DXGI_FORMAT_R32G32B32A32_{ FLOAT,SINT,UINT }。

⑩ DXGI_FORMAT_R32G32_{ FLOAT,SINT,UINT }。

⑪ DXGI_FORMAT_R32_{ SINT,UINT}。

⑫ DXGI_FORMAT_R8G8B8A8_{ SINT, SNORM, UINT, UNORM, UNORM_SRGB }。

⑬ DXGI_FORMAT_R8G8_ { SINT,SNORM,UINT,UNORM }。

如果 pD3DResource 类型不正确或已被注册，那么返回 cudaErrorInvalidResourceHandle。如果 pD3DResource 不能被注册，那么返回 cudaErrorUnknown。

返回值：cudaSuccess；cudaErrorInvalidDevice；cudaErrorInvalidValue；cudaErrorInvalidResourceHandle；cudaErrorUnknown

11. 错误处理

错误处理包含了运行时调用的最新错误和返回错误的消息字符串，有利于帮助开发者快速发现和定位程序中的错误，提高编程效率。

(1) cudaGetLastError

函数功能：返回运行时调用的最新错误。

调用形式：cudaError_t cudaGetLastError(void)

函数说明：返回同一主线程中运行时调用所返回的最新错误，并将其重置为 cudaSuccess。

返回值：cudaSuccess；cudaErrorInitializationError；cudaErrorLaunchFailure；cudaErrorPriorLaunchFailure；cudaErrorLaunchTimeout；cudaErrorLaunchOutOfResources；cudaErrorInvalidDeviceFunction；cudaErrorInvalidConfiguration；cudaErrorInvalidDevice；cudaErrorInvalidValue；cudaErrorInvalidDevicePointer；cudaErrorInvalidTexture；cudaErrorInvalidTextureBinding；cudaErrorInvalidChannelDescriptor；cudaErrorTextureFetchFailed；cudaErrorTextureNotBound；cudaErrorSynchronizationError；cudaErrorUnknown；cudaErrorInvalidResourceHandle；cudaErrorNotReady

（2）cudaGetErrorString

函数功能：返回错误中的消息字符串。

调用形式：const char * cudaGetErrorString (cudaError_t error)

返回值：指向 NULL 终止字符串的 char * 指针。

4.6　程序示例

本节通过例程简单介绍 CUDA 编程常识、常用的函数，并说明主机和设备之间如何传递参数、如何工作。

4.6.1　Hello World 实现

首先在主机端，建立一个.cpp 文件，使用如下程序，可以在屏幕上显示 Hello World，程序如下，运行的结果如图 4-54 所示。

```
int main( void )
{
    printf( "Hello World CPU\n" );
    getchar();
    return 0;
}
```

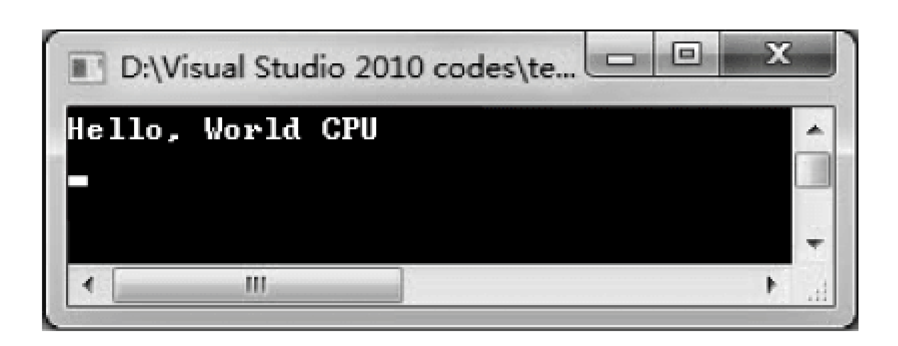


图 4-54　CPU 显示 Hello World

这段程序很简单，是在主机端实现的。现在需要将其进行适当的修改，使其可以在 GPU 中运行。在 GPU 上运行的函数，通常被称为核函数(kernel)，程序如下，运行结果如图 4-55 所示。

```
#include <iostream>
#include <cuda_runtime.h>

__global__ void kernel( void )
{
```

```
    printf( "Hello, World GPU\n" );
}

int main( void ) {
    kernel<<<1,1>>>();
    cudaDeviceSynchronize();
    getchar();
    return 0;
}
```



图 4-55　GPU 显示 Hello World

该程序运行的流程是：先运行 CPU 内的主函数，通过 kernel 函数开始调用 GPU 内的程序，在 GPU 内输出完 Hello World 之后，继续在 CPU 内运行程序至程序结束。

① kernel<<<1,1>>>()；

开辟了一个线程块中的一个线程，在这个线程中输出了 Hello World。因为没有数据传输进入 GPU，因此可以不用 cudaMalloc 开辟内存(显存)空间。

② cudaDeviceSynchronize()；

这个函数是 CUDA 内的同步函数，可以暂停 CPU 的进程，等待 GPU 内的函数都运行完成之后，继续向下运行 CPU 内的程序。

4.6.2　参数传递

前面的例程中没有将主机的参数传送给设备机，但实际应用中，必须将大量参数传递给 GPU，借助 GPU 高强度的速浮点计算能力，大大提升处理效率。本节给出一个将主机端的两个数字在设备端加和的程序，程序实现如下，实现的结果如图 4-56 所示。

```
#include <cuda_runtime.h>
#include <iostream>

__device__ int add_d( int a, int b ) {
    return a + b;
}

__global__ void add( int a, int b, int *c ) {
    *c = add_d( a, b );
}

int main( void )
{
    int c;
    int *ptr;
```

```
    cudaMalloc( (void * * )&ptr, sizeof(int) );
    add <<< 1,1 >>>( 1, 2, ptr );
    cudaMemcpy( &c, ptr, sizeof(int),cudaMemcpyDeviceToHost );
    printf( " 1 + 2 =  % d\n", c );
    cudaFree( ptr );
    getchar();
    return 0;
}
```



图 4-56　参数传递

这套程序的运行流程如下：

① 程序先从主函数进入，在 CPU 中定义了变量 c 和指针 ptr，同时通过 cudaMalloc 函数开辟了 GPU 中的内存。

② 将数字和指针一同传送到 GPU 中，在__global__下的 add 函数中运行。

③ add 函数并没有实现计算，而是继续调用同在 GPU 中处理的__device__，在__device__中通过 add_d 函数实现了计算，并将计算后的结果返回给__global__。

④ GPU 内程序运行完之后，GPU 通过 cudaMemcpy 将 GPU 内计算后的结果返回到 CPU 中。

⑤ 在屏幕上输出结果，并释放指针，程序运行结束。

该程序的计算过程虽然简单，不过通过几次调用可以帮助理解 CPU 和 GPU 之间如何进行数据传递，以及 GPU 内部的函数如何被调用。

需要注意的是：

① cudaMalloc()函数

cudaMalloc()是分配内存的函数，该函数类似于 C 语言中的 malloc()函数，不过该函数的功能不是在 CPU 中分配内存，而是在 GPU 中分配内存，这个内存也仅供 GPU 在计算时使用。

程序 cudaMalloc((void **)&ptr, sizeof(int))中，第一个参数类型是指针，指向用于保存新分配地址的变量；第二个参数表示分配内存的大小，因为是 int 型的参数，所以使用 sizeof (int)就可以直接分配好 int 型数据所需要的空间。函数的详细用法可参见 4.5.2 节。

但是 cudaMalloc 函数也存在着局限性，它本身为 CUDA C 语言，与标准的 C 语言还是有很大的不同，这种不同总结为以下四点：

a. cudaMalloc()分配的指针可以传递给设备上执行的函数。

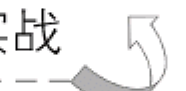
b. 设备中的程序可以使用 cudaMalloc()函数分配的指针进行内存的读/写操作。

c. cudaMalloc()分配的指针可以传递给主机上的执行函数。

d. 不能在主机程序中使用 cudaMalloc()分配的指针进行内存的读/写操作。

② cudaMemcpy()

cudaMemcpy()是数据传递函数，同样类似于标准 C 语言的 memcpy()，调用的方式也



很相似，但是 cudaMemcpy()仅限于 GPU 内的数据传递。需要注意的是最后一个参数 cudaMemcpyDeviceToHost，这个参数就是和标准 C 语言不同的地方，这个参数说明在运行时，源指针是一个设备指针，而目标指针是一个主机指针。当然有些时候也会使用 cudaMemcpyHostToDevice，它与之前是完全相反的含义，即源指针位于主机上，而目标指针位于设备上。

cudaMemcpyDeviceToDevice 指源指针和目标指针都位于设备机上。

cudaMemcpyHostToHost 指源指针和目标指针都在主机端，当然，如果两者都在主机端，则可以直接调用标准 C 语言中的 memcpy()函数实现相同的功能。

③ cudaFree()

cudaFree()是释放指针的函数，也是类似于标准 C 语言中的 free()函数，而且与 free()的区别同上述 cudaMalloc()与 memcpy()的区别类似，free()只能对主机端的指针进行释放。在主机端定义的指针如果曾经被调用到过设备端，就需要用 cudaFree()函数来释放这个指针。

④ __global__和__device__

限定符__global__声明的函数为内核。此类函数在设备上执行，仅可通过主机才能进行调用。

限定符__device__声明的函数在设备上执行，仅可通过设备调用。

4.6.3 同步函数

同步函数是开发 CUDA 程序所需的重要函数之一。很多复杂的程序需要等待所有设备端都执行完后再继续进行主机端的程序，这时就需要同步函数暂时阻塞主机端程序的运行，直到所有设备端的程序执行完毕。

当数据从 CPU 端传送到 GPU 端时，GPU 开始处理数据，但是在不使用同步函数的情况下，CPU 端也将继续运行程序，直到需要 GPU 中的数据时才停下，若不需要 GPU 中的数据，则直接运行结束。如图 4-57 所示为不使用同步函数的情况下，CPU 与 GPU 执行任务的顺序。CPU 端在执行任务 2 时将数据传递给 GPU，GPU 开始执行任务 A 和任务 B，因为没有使用同步函数，所以 CPU 将不会等待 GPU 的结果，继续执行任务 3 和任务 4。直到任务 4 快完成的时候，GPU 处理完成，将结果返回给了 CPU，当然也有可能计算机在完成任务 4 之后，GPU 仍然没有工作完成。因为 GPU 执行完成的时间不可控，所以会给编程带来很大的麻烦。

本节将对前面的程序进行适当修改，在不使用同步函数的情况下，分别在主机和设备端上添加几个输出，来查看主机端和设备端的执行顺序。修改后的程序如下所示：

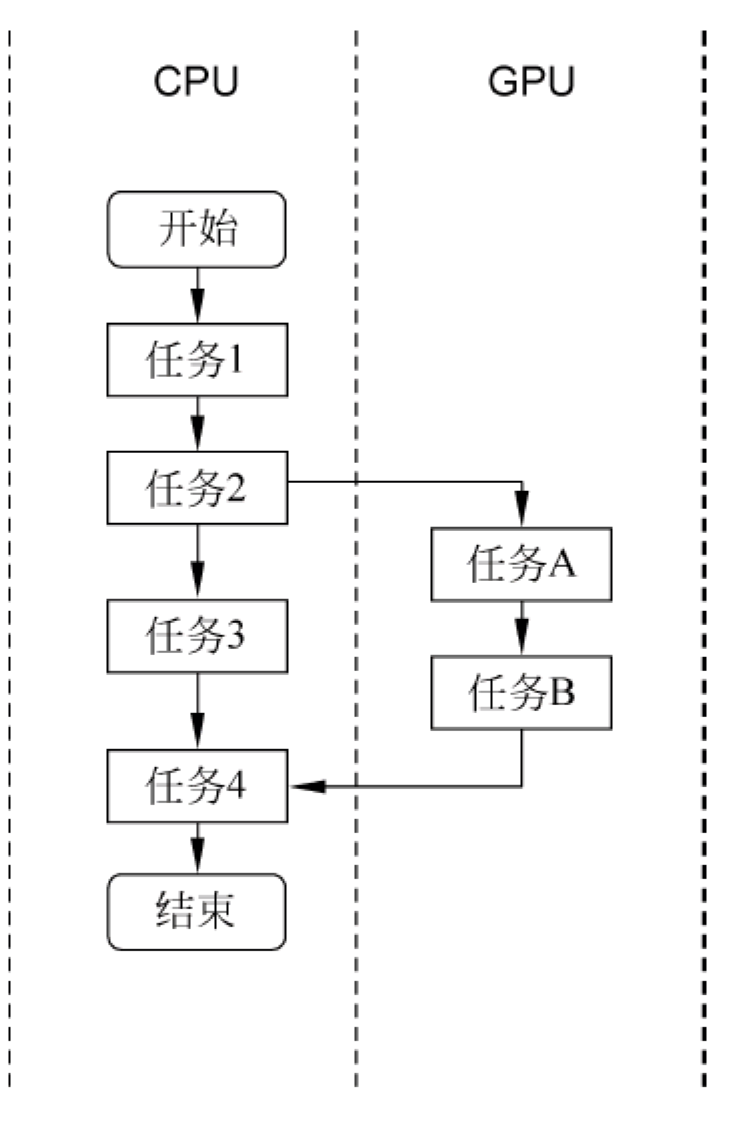


图 4-57　执行顺序

```
#include <cuda_runtime.h>
#include <iostream>
```

```
__device__ int add_d( int a, int b ) {
    printf("Hello_World_3\n");
    return a + b;
}

__global__ void add( int a, int b, int *c ) {
    printf("Hello_World_2\n");
    *c = add_d( a, b );
    printf("Hello_World_4\n");
}

int main( void )
{
    int c;
    int *ptr;
    cudaMalloc( (void**)&ptr, sizeof(int) );
    printf("Hello_World_1\n");
    add<<<1,1>>>( 1, 2, ptr );
    printf("Hello_World_5\n");
    cudaMemcpy( &c, ptr, sizeof(int),cudaMemcpyDeviceToHost );
    printf("Hello_World_6\n");
    printf( " 1 + 2 = %d\n", c );
    cudaFree( ptr );
    getchar();
    return 0;
}
```

程序运行结果如图 4-58 所示。



图 4-58 参数传递程序运行结果

运行程序可以采用单步调试的方法,但是单步调试很难把过程表示出来,因此采用输出几个 Hello_World 和数字的方式来查看实际应用中主机和设备之间执行的顺序。通过输出的顺序,可以看出主机端使用 add()函数将数据传递到设备端之后,主机端仍然在向下执行程序,在设备端输出 Hello_World_2 之前主机端已输出 Hello_World_5。

为了保证线程的同步,CUDA 推出了线程同步函数:

cudaDeviceSynchronize()函数可以暂时阻塞 CPU 内的程序,待所有设备端内的线程都运行完之后再继续主机端的程序。这样,可以保证设备端内的线程同步,避免很多不必要的麻烦。

与其相似的还有另外两个同步函数 cudaThreadSynchronize()和 cudaStreamSynchronize()函数。

cudaThreadSynchronize()的使用方法和效果与 cudaDeviceSynchronize()基本相同,但它是一种较旧的表达方式,并且可控性不强,所以不推荐使用。

cudaStreamSynchronize()函数接受一个 stream ID,它将阻止 CPU 执行直到 GPU 端完成对应 stream ID 的所有 CUDA 任务,但其他 stream 中的 CUDA 任务可能执行完毕也可能没有执行完。

在上述程序中加上同步函数的程序如下:

```
#include <cuda_runtime.h>
#include <iostream>

__device__ int add_d( int a, int b ) {
    printf("Hello_World_3\n");
    return a + b;
}

__global__ void add( int a, int b, int *c ) {
    printf("Hello_World_2\n");
    *c = add_d( a, b );
    printf("Hello_World_4\n");
}

int main( void )
{
    int c;
    int *ptr;
    cudaMalloc( (void**)&ptr, sizeof(int) );
    printf("Hello_World_1\n");
    add<<<1,1>>>( 1, 2, ptr );
    cudaDeviceSynchronize();
    printf("Hello_World_5\n");
    cudaMemcpy( &c, ptr, sizeof(int),cudaMemcpyDeviceToHost );
    printf("Hello_World_6\n");
    printf( " 1 + 2 = %d\n", c );
    cudaFree( ptr );
    getchar();
    return 0;
}
```

同步函数运行后的结果就是正常的 CPU—GPU—CPU 的流程了,如图 4-59 所示。不是每次调用 GPU 都必须使用同步函数,适当利用 GPU 计算的时间让 CPU 适度地进行一些简单计算也可以提高整体计算效率。当需要同步时,CPU 可以在将数据传递到 GPU 之后,直接使用 cudaMemcpy()函数等待 GPU 的数据传送回来以实现同步。



图 4-59 同步后的结果

4.7　线程层次

前面的介绍都没有牵涉较多数据的计算，在使用 CUDA 时也都只开启了一个线程块中的一个线程，但是在计算数据量较大时就必须开启多个线程。本节将从线程层次的角度通过例子来逐步介绍如何使用 GPU 去做并行计算。其中 4.7.1 节是对核函数和线程层次的总述，后面几节则通过例程给出说明。

4.7.1　核函数调用和线程层次介绍

GPU 中的每一个线程都有其特定的线程 ID，通过线程 ID 可以明确地给每一个线程分配任务，线程的 ID 信息由变量 threadIdx 和 blockIdx 给出。

在 GPU 的结构中，线程块(block)和每个块中的线程(thread)都不是一维的。线程块可以是一维的、二维的，也可以是三维的，同样对于每个线程块中的线程也可以是一维、二维、三维。如图 4-60 所示是二维线程块和二维线程的例子。

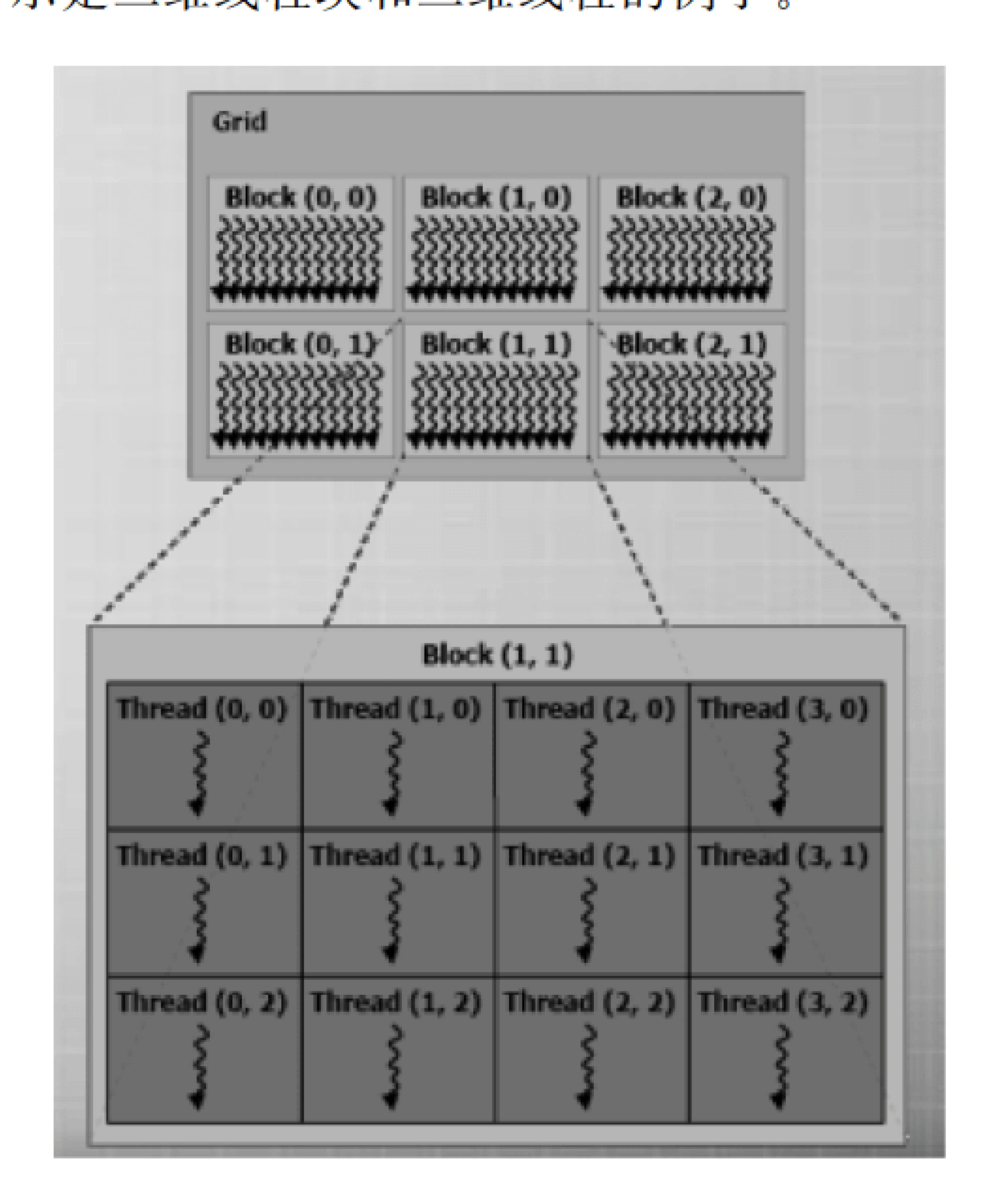


图 4-60　二维线程块和线程

1. 核函数调用

在前面的例程中，包含了核函数的调用。核函数的表述形式为：

```
Func <<< Dg,Db,Ns,s >>>(parameter);
```

其中：

(1) Dg 的类型为 dim3，指定线程网格的维度和大小，线程网格是二维的，但是为了能够在 dim3 类型中使用，仅有 x 维度和 y 维度上有数据，所以 Dg. x×Dg. y 等于所启动线程块的数量，Dg. z 设定为 0。在使用时如果不用 dim3 类型的变量，直接输入 int 型变量，则认

为是一维变量,Dg. y 和 Dg. z 也会自动设置为 0。

(2) Db 的类型为 dim3,指定线程块的维度和大小,Db. x * Db. y * Db. z 等于每个线程块的线程数量。在使用时如果不用 dim3 类型的变量,直接输入 int 型变量,则认为是一维变量,Db. y 和 Db. z 也会自动设置为 0。

(3) Ns 的类型为 size_t,指定为此调用动态分配的共享存储器的大小(除静态分配的存储器),这些动态分配的存储器可供声明为 extern 的数组使用,Ns 是一个可选参数,默认值为 0。

(4) s 的类型为 cudaStream_t,指定相关流;s 是一个可选参数,默认值为 0。

2. 一维线程索引

为了在使用时可以明确地调用每一个线程,CUDA C 语言定义了 threadIdx 变量专门用来进行线程索引。

threadIdx 是 CUDA C 语言的内建变量,通常用一个三维数组来表示。使用三维数组的方便之处在于可以简明地表示一维、二维和三维线程索引,进而方便地表示一维、二维和三维线程块(thread block)。这样,无论是数组、矩阵还是体积的计算,都可以很容易地使用 CUDA 进行运算。

线程的索引与线程 ID 之间存在着直接的换算关系,对于一个索引为(x, y, z)的线程来说:

① 如果线程块是一维的,则线程 ID=x。

② 如果线程块是二维的,假设块尺寸为(Dx,Dy),那么线程 ID=x+y * Dx。

③ 如果线程块是三维的,假设其尺寸为(Dx,Dy,Dz),那么线程 ID=x+y * Dx+z * Dx * Dy。

3. 二维、三维线程索引

使用过程中线程块可以设定为二维的或三维的,这时就需要从较高的维度来查找线程的数量。所以 CUDA 定义了如下几个参数来索引线程:

① gridDim——代表线程格的尺寸,gridDim. x 为 x 轴尺寸,gridDim. y 为 y 轴尺寸、gridDim. z 为 z 轴尺寸。

如图 4-60 所示,图中的线程格中包含的 gridDim. x=3,gridDim. y=2,gridDim. z=1。

② blockDim——代表线程块的尺寸,blockDIm. x 为 x 轴尺寸,blockDIm. y 为 y 轴尺寸,blockDIm. z 为 z 轴尺寸。

用图 4-60 中的 Block(1,1)来说,其内部的线程尺寸为:blockDim. x=4,blockDim. y=3,blockDim. z=1。

③ blockIdx——代表线程块在线程格中的索引值,也可以理解为一个线程块在其线程格中的坐标,blockIdx. x 为 x 轴上的坐标,blockIdx. y 为 y 轴上的坐标,blockIdx. z 为 z 轴上的坐标。

仍使用图 4-60,Block(1,1)的索引值为:blockIdx. x=1,blockIdx. y=1,blockIdx. z=0。通常情况下二维的就不需要再写 blockIdx. z=0,默认其为 0;同样,一维的也不需要特意写出 blockIdx. y 和 blockIdx. z。

④ threadIdx——线程索引,在前面已经做过简单的介绍,此处不再赘述。

4. thread 表达式

在调用核函数时，一般会使用一维或者二维的线程块和线程，虽然线程格和线程块的划分方式不同（一维、二维、三维），但是在 CUDA 程序进行的任意时刻，每一个线程的 threadID 都是唯一标识且不可改变的。下面给出 threadID 的完整表达式：

① 主函数定义 grid 为一维，block 为一维：

```
__device__ int kernel_G1_B1()
{
    int threadID = blockIdx.x * blockDim.x
                 + threadIdx.x;
    ...
```

② 主函数定义 grid 为一维，block 为二维：

```
__device__ int kernel_G1_B2()
{
    int threadID = blockIdx.x * blockDim.x * blockDim.y
       + threadIdx.y * blockDim.x
       + threadIdx.x;
    ...
}
```

③ 主函数定义 grid 为一维，block 为三维：

```
__device__ int kernel_G1_B3()
{
    int threadID = blockIdx.x * blockDim.x * blockDim.y * blockDim.z
       + threadIdx.z * blockDim.y * blockDim.x
       + threadIdx.y * blockDim.x
       + threadIdx.x;
    ...
}
```

④ 主函数定义 grid 为二维，block 为一维：

```
__device__ int kernel_G1_B1()
{
    int blockID = blockIdx.y * gridDim.x + blockIdx.x;
    int threadID = blockID * blockDim.x + threadIdx.x;
    ...
}
```

⑤ 主函数定义 grid 为二维，block 为二维：

```
__device__ int kernel_G2_B2()
{
    int blockID = blockIdx.y * gridDim.x + blockIdx.x;
    int threadID = blockID * blockDim.x * blockDim.y //block 乘进来是为了确认小块
       + threadIdx.y * blockDim.x
       + threadIdx.x;
    ...
}
```

⑥ 主函数定义 grid 为二维，block 为三维：

```
__device__ int kernel_G2_B3()
{
    int blockID = blockIdx.y * gridDim.x + blockIdx.x;
    int threadID = blockID * blockDim.x * blockDim.y * blockDim.z
        + threadIdx.z * blockDim.x * blockDim.y
        + threadIdx.y * blockDim.x
        + threadIdx.x;
    ...
}
```

⑦ 主函数定义 grid 为三维，block 为一维：

```
__device__ int kernel_G3_B1()
{
    int blockID = blockIdx.z * gridDim.x * gridDim.y
        + blockIdx.y * gridDim.x
        + blockIdx.x;
    int threadID = blockID * blockDim.x + threadIdx.x;
    ...
}
```

⑧ 主函数定义 grid 为三维，block 为二维：

```
__device__ int kernel_G3_B2()
{
    int blockID = blockIdx.z * gridDim.x * gridDim.y
        + blockIdx.y * gridDim.x
        + blockIdx.x;
    int threadID = blockID * blockDim.x * blockDim.y
        + threadIdx.y * blockDim.x
        + threadIdx.x;
    ...
}
```

⑨ 主函数定义 grid 为三维，block 为三维：

```
__device__ int kernel_G3_B3()
{
    int blockID = blockIdx.z * gridDim.x * gridDim.y
        + blockIdx.y * gridDim.x
        + blockIdx.x;
    int threadID = blockID * blockDim.x * blockDim.y * blockDim.z
        + threadIdx.z * blockDim.x * blockDim.y
        + threadIdx.y * blockDim.x
        + threadIdx.x;
}
```

⑩ 同样维度的线程也可以有不同的表达方式，最常用的就是二维线程格和二维线程块，这种写法常常在处理图像数据时使用，如下所示：

```
__device__ int kernel_G2_B2()
```

```
{
        const int x = blockIdx.x * blockDim.x + threadIdx.x;
        const int y = blockIdx.y * blockDim.y + threadIdx.y;
}
```

①～⑨的线程索引都是使用一个 threadID 变量来得到确定的线程，方式⑩是使用两个变量来确定线程的位置。

举例如下：

在 GPU 中共开启了 4×4 个线程块，每个线程块中开启了 3×3 个线程，现在需要找到坐标为(7,7)的线程，如图 4-61 所示。其中 blockIdx. x * blockDim. x 为线程块中的索引，其取值为 2×3。之后再加上每一个线程的索引 threadIdx. x，本次的取值为 1。所以，x 的取值为 7；同理，y 的取值也为 7，这样就可以实现二维线程格和二维线程块的索引。

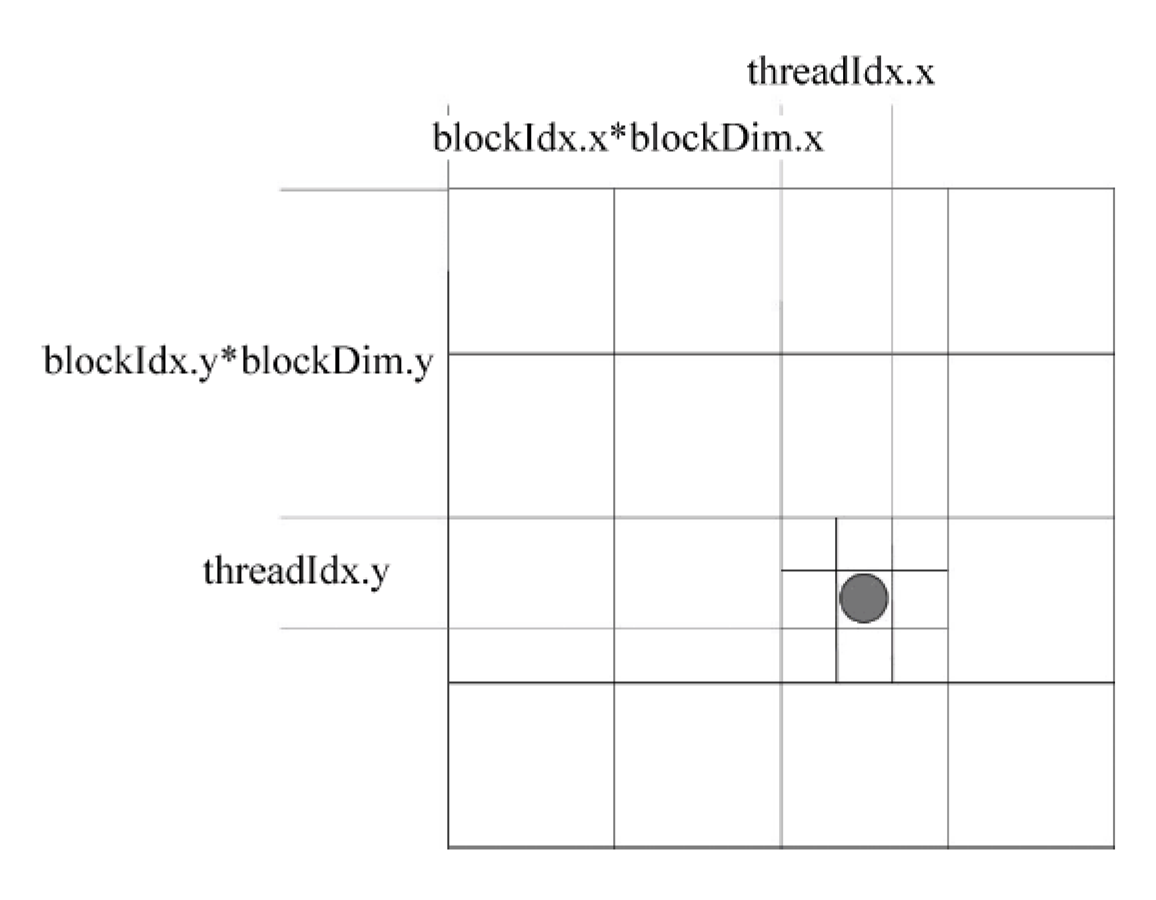


图 4-61　线程索引

4.7.2　矢量求和

在 4.7.1 节中，在主函数调用的过程中使用了：

```
add<<<1,1>>>( 1, 2, ptr );
```

第一个参数表示的是在本次运行的过程中启动的线程块数量，没有使用 dim3 型变量，而是使用了 int 型变量，表示启动一个线程块，第二个参数表示在每一个线程块中启动了一个线程。

如果需要计算 10 个数据，则需要启动 10 个线程，可以采用如下两种方式来启动这些线程：

```
Kernel<<<10,1>>>();
```

第一种表示启动了 10 个线程格，每个线程格启动 1 个线程，需要使用 blockIdx. x 来进行索引。

```
Kernel<<<1,10>>>();
```

第二种表示启动了 1 个线程格，在这个线程格中启动了 10 个线程，整体也是启动 10 个线

程，这种启动方式需要用 threadIdx. x 来进行索引。

如果需要编写较大的程序，就必须启用较多的线程，仅需把<<<、>>>中的值适当地调大就可以；但需要注意的是，在启动线程块组时，数组的每一个维度的最大数量都不能超过 65 535，这是硬件的限制，超过了这个限制，程序就将运行失败。下面用 10 个数字相加的程序作为例子来进行讲解。该程序将数组 a[10]和 b[10]中对应脚标位置的数据相加并将结果保存在 c[10]中。

例程如下：

```
#include <cuda_runtime.h>
#include <iostream>
#include <device_launch_parameters.h>

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    if (tid < 10)
        c[tid] = a[tid] + b[tid];
}

int main( void ) {
    int a[10], b[10], c[10];
    int *dev_a, *dev_b, *dev_c;

    cudaMalloc( (void**)&dev_a, 10 * sizeof(int) );
    cudaMalloc( (void**)&dev_b, 10 * sizeof(int) );
    cudaMalloc( (void**)&dev_c, 10 * sizeof(int) );

    for (int i = 0; i < 10; i++) {
     a[i] = i;
     b[i] = i + 2;
    }

    cudaMemcpy( dev_a, a, 10 * sizeof(int),cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, 10 * sizeof(int),cudaMemcpyHostToDevice );

    add<<<1,10>>>( dev_a, dev_b, dev_c );
    cudaMemcpy( c, dev_c, 10 * sizeof(int),cudaMemcpyDeviceToHost );

    for (int i = 0; i < 10; i++)
    {
       printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    getchar();
    return 0;
}
```

运行结果如图 4-62 所示。

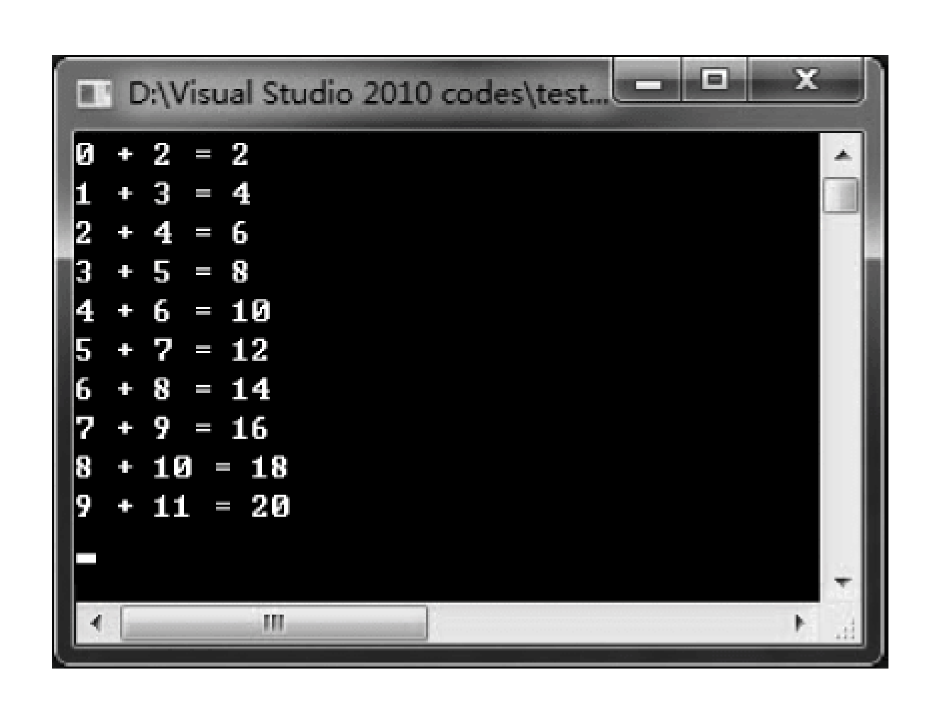


图 4-62　运算结果

给 int 型的变量赋予索引值，这个变量也就是前面说到的线程索引 threadID，通过它就可以对应找到设备端与之相匹配的线程，给每个线程赋值。设备端在得到这个指令时，每个线程相当于得到了如下的指令：

第一个线程，

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < 10)
       c[tid] = a[tid] + b[tid];   //把数组的第一位对应相加
}
```

第二个线程，

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < 10)
       c[tid] = a[tid] + b[tid];   //把数组的第二位对应相加
}
```

第三个线程，

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < 10)
       c[tid] = a[tid] + b[tid];   //把数组的第三位对应相加
}
…
```

第十个线程：

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = 9;
    if (tid < 10)
       c[tid] = a[tid] + b[tid];   //把数组的第十位对应相加
}
```

每一个线程得到的数据是不同的，因此可以使 10 个线程同时计算 10 组不同的数据，最

后一同从设备端返回给主机端。

当然，也可以采取使用线程块的方式来分配线程，将上述程序中的add()函数修改为：

```
add<<<10,1>>>( dev_a, dev_b, dev_c );
```

将设备端的程序修改为：

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;
    if (tid < 10)
      c[tid] = a[tid] + b[tid];
}
```

这样可以调用 10 个线程块，每个线程块中启动一个线程进行计算。这种方式和使用一个线程块中 10 个线程的方式有一定的区别。调用线程块的方式叫做粗粒度并行，而调用一个线程块内的 10 个线程就叫做细粒度并行。通常情况下，细粒度并行的效率更高，运行的时间更短一些。

4.7.3 数据较多的矢量求和

在前面的例子中，使用的是长度为 10 的数据，但在实际应用中，很容易出现数据量较大，在硬件限制下无法保证一个线程对应一个数据处理操作。因此，需要适当修改核函数的索引方式和核函数的调用方式以避免这个情况。

如果需要计算两个长度为 1024 的数组对应的和，可以采用启动 15 个线程块，并且每一个线程块仅启动一个线程的方式，对应的程序如下：

```
#include <cuda_runtime.h>
#include <iostream>
#include <device_launch_parameters.h>

#define N 1024
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += gridDim.x;
    }
}

int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );
```

```
    for (int i = 0; i < N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }

    cudaMemcpy( dev_a, a, N * sizeof(int),cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int),cudaMemcpyHostToDevice );

    add <<< 15,1 >>>( dev_a, dev_b, dev_c );

    cudaMemcpy( c, dev_c, N * sizeof(int),cudaMemcpyDeviceToHost );

    for (int i = 0; i < N; i++) {
        if ((a[i] + b[i]) == c[i]) {
            printf( "%d + %d = %d\n", a[i], b[i], c[i] );
        }
    }

    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    free( a );
    free( b );
    free( c );

    getchar();
    return 0;
}
```

程序运行的结果如图 4-63 所示。

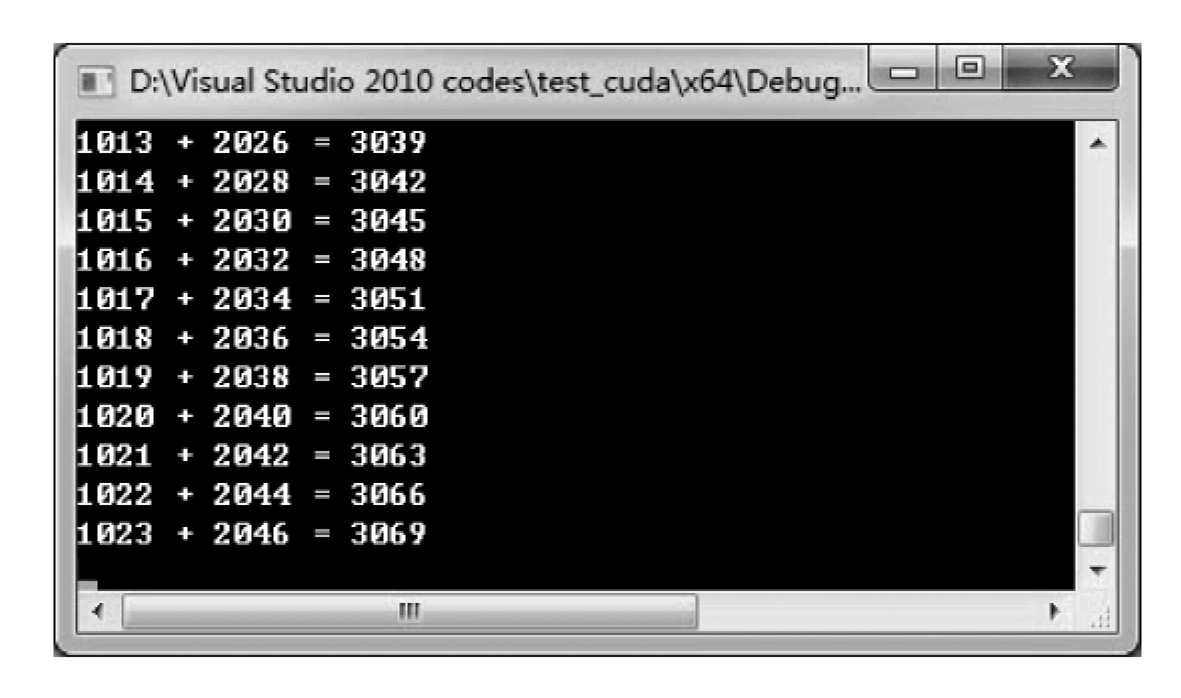
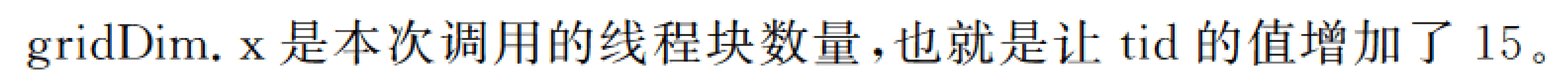


图 4-63　1024 个数据求和结果

这套程序跟上一套程序相比改动很小，只是增大了起始的数据量，并且限定了线程数量。这套程序在主机端的运行部分仍然是给两个数组赋值并将数据传送到设备端。而设备端的程序使用了循环语句来让所有的线程一次一次计算，最终得出结果。

① 在第一次运行设备端 while()循环时，tid 的值是 0～14，也就是说，使用了 15 个线程，分别计算了数组中的前 15 个数据。之后运行

```
tid + = gridDim.x;
```



gridDim.x 是本次调用的线程块数量，也就是让 tid 的值增加了 15。

② 第二次运行 while()循环时，仍然是原本的 15 个线程，但是在计算时 tid 的值发生了改变，这些线程在第二次循环时，计算的是数组中第 15～29 的数据。就这样依次循环下去。

③ 在最后一次循环时，while 已经循环了 67 次，在进行第 68 次计算时，tid += gridDim.x 的取值为 1020～1034。因为有 tid < N 的限制，所以很明显，不是所有的 tid 都可以进入到下一次循环中，仅有 tid < N 的部分是可以进入下一次循环。也就是说，在最后一次计算时剩下的 a[1020]、a[1021]、a[1022]、a[1023]、a[1024]和 b[1020]、b[1021]、b[1022]、b[1023]、b[1024]，将由 tid 取值小于 N 的 5 个线程来进行相加计算。这也是 CUDA 自身的一个保护功能，可以在保证计算资源足够的情况下，尽可能地减少不必要的开销。

4.7.4 不同维度线程索引

在前面的索引中，使用的线程块和线程都是低维度的，本节将给出启动高维线程块和线程的程序以及解析。

1. 一维线程块和一维线程

本节的例子是在之前程序的基础上，分别启动了 64 个一维线程块和 64 个一维线程，用来计算两个长度为 32 768 的数组相加并将结果存在第三个数组中。

程序如下：

```
#include <cuda_runtime.h>
#include <iostream>
#include <device_launch_parameters.h>
#define N (32 * 1024)

__global__ void add( int *a, int *b, int *c )
{
    int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    while (threadID < N) {
        c[threadID] = a[threadID] + b[threadID];
        threadID += blockDim.x * gridDim.x;
    }
}

int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;

//////////在 CPU 上开辟内存
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

/////////在 GPU 上开辟内存
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );
```

```
    for (int i = 0; i < N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }

//////////CPU 向 GPU 传递数据
    cudaMemcpy( dev_a, a, N * sizeof(int),cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int),cudaMemcpyHostToDevice );

    add<<<64,64>>>( dev_a, dev_b, dev_c );

/////////数据传送回 CPU
    cudaMemcpy( c, dev_c, N * sizeof(int),cudaMemcpyDeviceToHost );

    for (int i = 0; i < N; i++) {
        if ((a[i] + b[i]) == c[i]) {
            printf( " %d + %d = %d\n", a[i], b[i], c[i] );
        }

    }
/////////释放 GPU 指针
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
/////////释放 CPU 指针
    free( a );
    free( b );
    free( c );

    getchar();
    return 0;
}
```

程序运行的结果如图 4-64 所示。

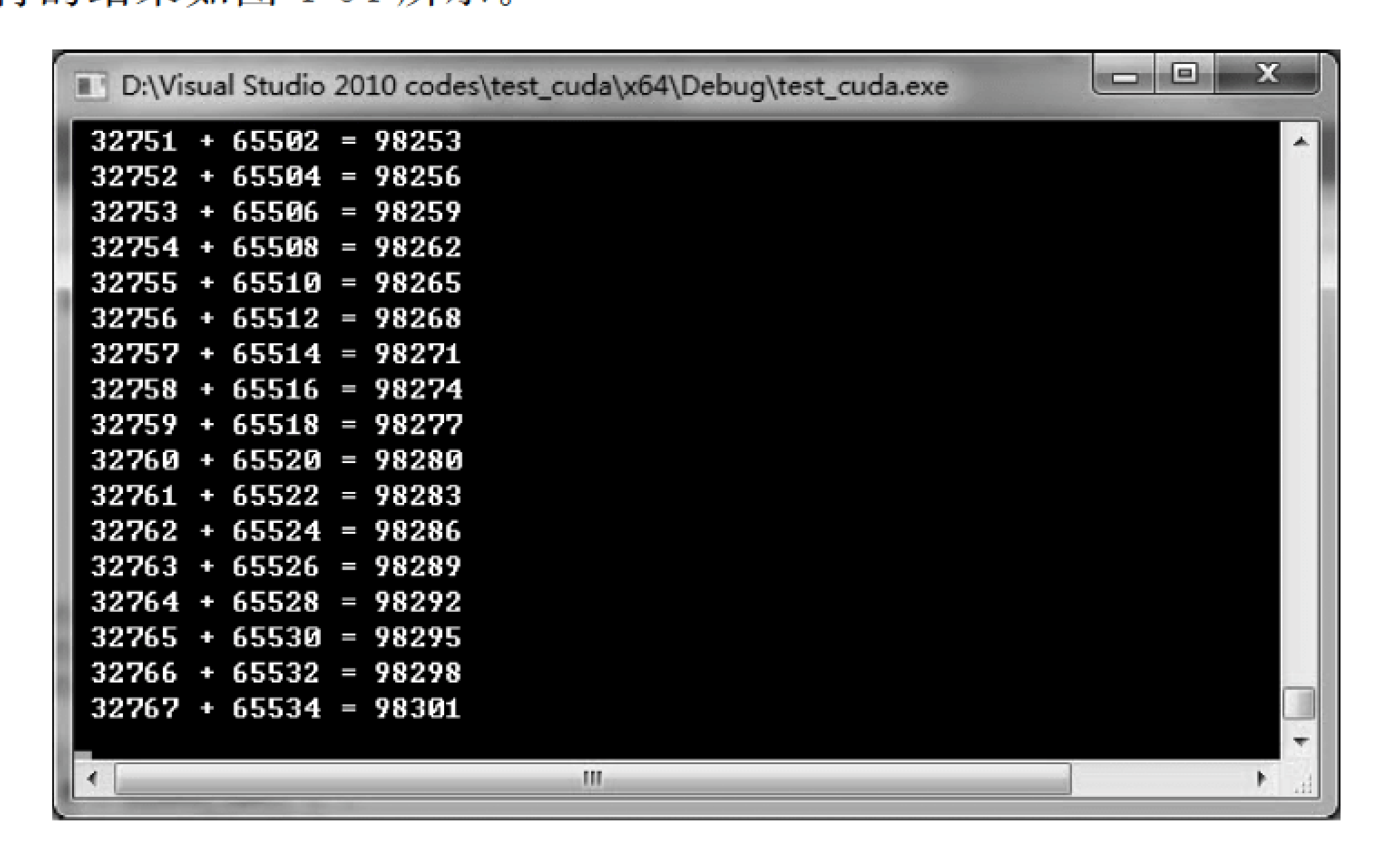


图 4-64 一维计算结果

相信有了前面的基础，看懂这段程序不是难事，需要注意的是，每次 threadID 增加的数量就是每次启动线程数量的总数。因为本次是一维的，所以 blockDim. x * gridDim. x 的总体数量就是 64×64 启动的总体，即 4096 个线程。

2. 二维线程块和二维线程

在上面程序的基础上，将每次启动的线程块和线程都修改成为二维的，对应的程序如下：

```
#include <cuda_runtime.h>
#include <iostream>
#include <device_launch_parameters.h>

#define N (32 * 1024)

__global__ void add( int *a, int *b, int *c ) {

    int blockID = blockIdx.y * gridDim.x + blockIdx.x;
    int threadID = blockID * blockDim.x * blockDim.y
        + threadIdx.y * blockDim.x
        + threadIdx.x;

    while (threadID < N) {
        c[threadID] = a[threadID] + b[threadID];

        threadID += gridDim.x * gridDim.y * blockDim.x * blockDim.y;
    }
}

int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;

//////////在 CPU 上开辟内存
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

/////////在 GPU 上开辟内存
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );

    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }

//////////向 GPU 传递数据
    cudaMemcpy( dev_a, a, N * sizeof(int),cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int),cudaMemcpyHostToDevice );
```

```
    dim3 grids(8,8);
    dim3 threads(8,8);
    add <<< grids,threads >>>( dev_a, dev_b, dev_c );

/////////数据传送回主机端
    cudaMemcpy( c, dev_c, N * sizeof(int),cudaMemcpyDeviceToHost );

    for (int i = 0; i < N; i++) {
        if ((a[i] + b[i]) == c[i]) {
            printf( " %d + %d = %d\n", a[i], b[i], c[i] );
        }

    }
/////////释放 GPU 指针
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
/////////释放 CPU 指针
    free( a );
    free( b );
    free( c );

    getchar();
    return 0;
}
```

程序运行的结果如图 4-65 所示。

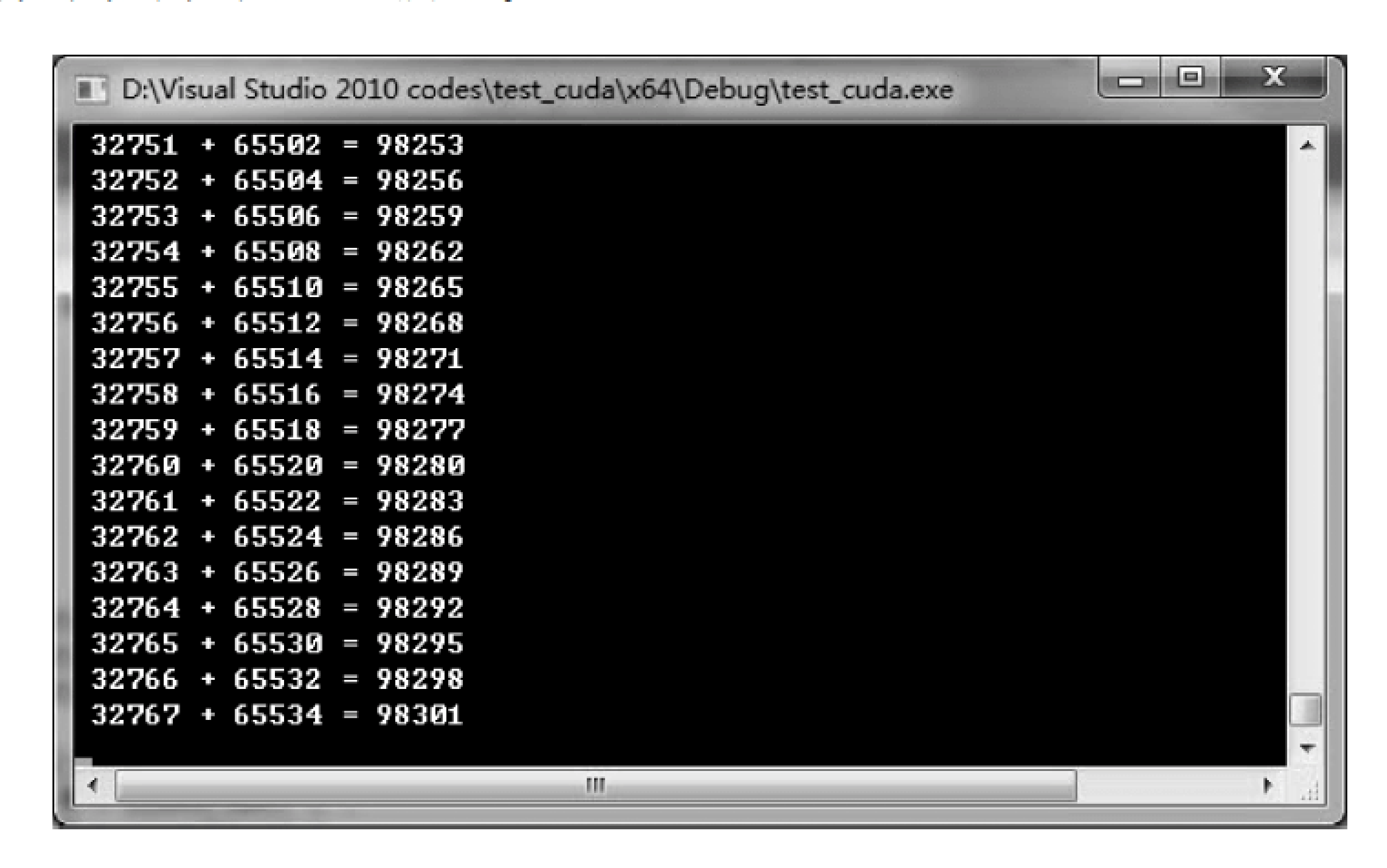


图 4-65　二维计算结果

程序简析如下：

① 使用了 dim3 类型的变量来调用内核函数。其中：

```
dim3 grids(8,8);
dim3 threads(8,8);
add <<< grids,threads >>>( dev_a, dev_b, dev_c );
```

grids 启动了 8×8 的二维线程块，threads 表示在每一个线程块中启动了 8×8 的线程。add 函数将两个 dim3 类型的变量传递给了核函数。

② 核函数中的线程索引部分：

```
int blockID = blockIdx.y * gridDim.x + blockIdx.x;
int threadID = blockID * blockDim.x * blockDim.y
    + threadIdx.y * blockDim.x
    + threadIdx.x;
```

设定了两个 int 类型的值，第一个 blockID 是将所有的线程格索引列出来，而 threadID 则是在 blockID 的基础上，将所有的线程都做好对应的索引。

③ 迭代：

```
threadID + = gridDim.x * gridDim.y * blockDim.x * blockDim.y;
```

在每一个线程都得到一对数据并进行相加之后，还需要继续进行迭代来确保所有的数据都进行过计算，即每次在 threadID 中递增所有线程的数量。

gridDim.x * gridDim.y 表示启动的所有的线程块数量，即 8×8 的线程块；而 blockDim.x * blockDim.y 则表示每个线程块中启动线程的数量，即 8×8 的线程，所以每次递增的数量即为 4096 个。

3. 三维线程块和三维线程

虽然在实际应用中很少会使用这么高维的线程索引，但是为了方便讲解，还是启用三维的线程块和三维的线程来实现这个算法，对应的程序如下所示：

```
#include <cuda_runtime.h>
#include <iostream>
#include <device_launch_parameters.h>

#define N (32 * 1024)

__global__ void add( int *a, int *b, int *c ) {

    int blockID = blockIdx.z * gridDim.x * gridDim.y
        + blockIdx.y * gridDim.x
        + blockIdx.x;
    int threadID = blockID * blockDim.x * blockDim.y * blockDim.z
        + threadIdx.z * blockDim.x * blockDim.y
        + threadIdx.y * blockDim.x
        + threadIdx.x;

    while (threadID < N) {
        c[threadID] = a[threadID] + b[threadID];
        threadID += gridDim.x * gridDim.y * gridDim.z * blockDim.x * blockDim.y * blockDim.z;
    }
}

int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;

//////////在 CPU 上开辟内存
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
```

```
    c = (int*)malloc( N * sizeof(int) );

/////////在 GPU 上开辟内存
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );

    for (int i = 0; i < N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }

//////////向 GPU 传递数据
    cudaMemcpy( dev_a, a, N * sizeof(int),cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int),cudaMemcpyHostToDevice );

    dim3 grids(4,4,4);
    dim3 threads(4,4,4);
    add<<<grids,threads>>>( dev_a, dev_b, dev_c );

/////////数据传送回主机端
    cudaMemcpy( c, dev_c, N * sizeof(int),cudaMemcpyDeviceToHost );

    for (int i = 0; i < N; i++) {
        if ((a[i] + b[i]) == c[i]) {
            printf( " %d + %d = %d\n", a[i], b[i], c[i] );
        }
    }
/////////释放 GPU 指针
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
/////////释放 CPU 指针
    free( a );
    free( b );
    free( c );

    getchar();
    return 0;
}
```

程序运行的结果如图 4-66 所示。

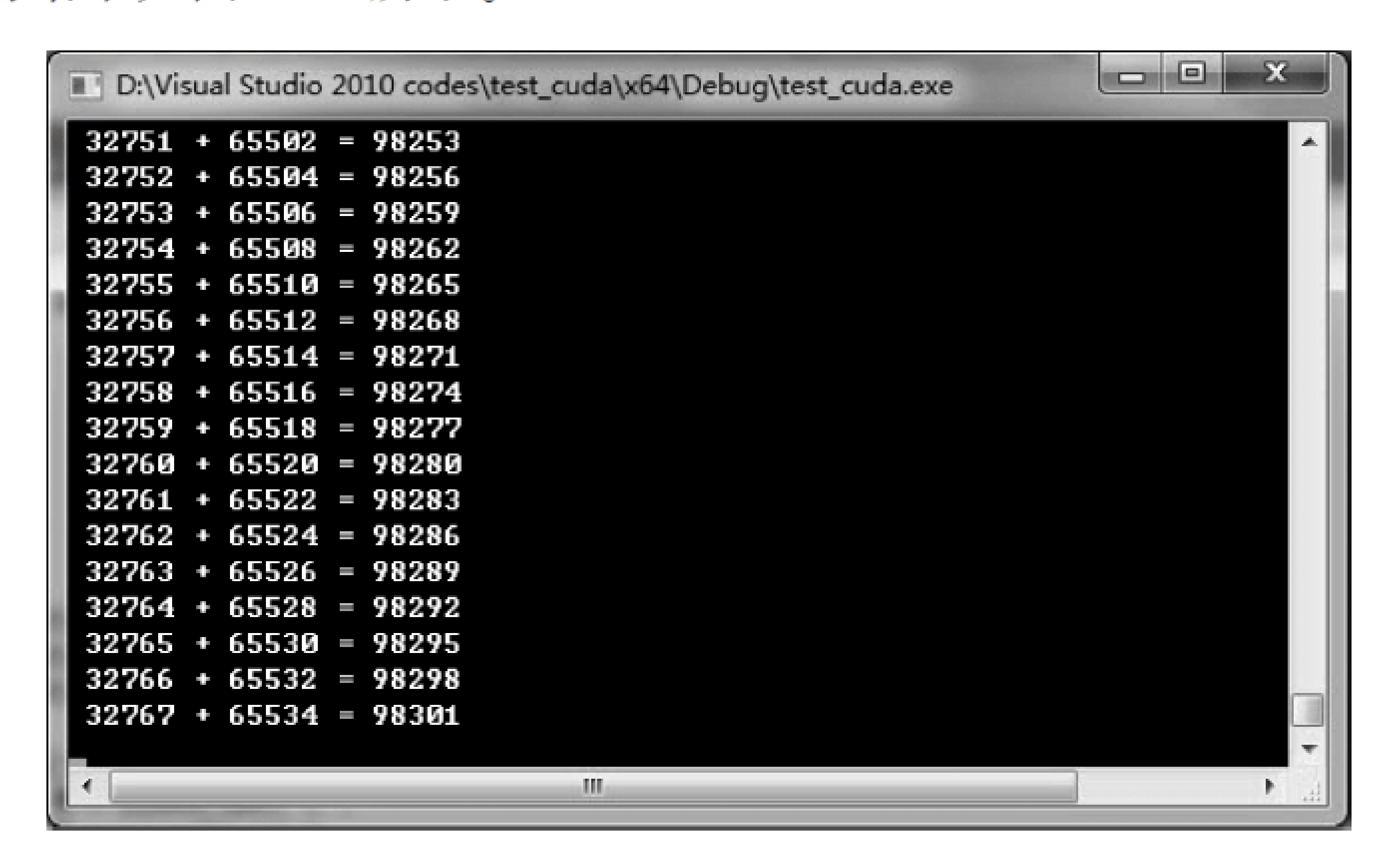


图 4-66　三维计算结果

程序简析如下：

① 这套程序使用了 dim3 类型的变量来调用内核函数。其中：

```
dim3 grids(4,4,4);
dim3 threads(4,4,4);
add <<< grids,threads >>>( dev_a, dev_b, dev_c );
```

grids 启动了 4×4×4 的三维线程块，threads 表示在每一个线程块中启动了 4×4×4 的线程。add 函数将两个 dim3 类型的变量传递给了核函数。

② 核函数中的线程索引部分：

```
int blockID = blockIdx.z * gridDim.x * gridDim.y
        + blockIdx.y * gridDim.x
        + blockIdx.x;
int threadID = blockID * blockDim.x * blockDim.y * blockDim.z
        + threadIdx.z * blockDim.x * blockDim.y
        + threadIdx.y * blockDim.x
        + threadIdx.x;
```

设定了两个 int 类型的值，blockID 将所有线程格索引列出来，而 threadID 则在 blockID 的基础上，将所有的线程都对应好相应的索引。

③ 迭代：

```
threadID + = gridDim.x * gridDim.y * gridDim.z * blockDim.x * blockDim.y * blockDim.z;
```

迭代是指每次循环叠加上启动的所有线程数量，启动的总线程块数量为 gridDim. x * gridDim. y * gridDim. z，即 4×4×4，一共 64 个线程块。每个线程块启动线程数量为 blockDim. x * blockDim. y * blockDim. z，即 4×4×4，一共 64 个线程，所以总体来看也是启动了共计 4096 个线程。

可以看出，上面使用三种不同维度的线程块和线程来解决同一问题，得到的结果是相同的。

4.8 GPU 的存储器

计算机主机在执行任务时，数据需要在内存中进行计算。CUDA 程序在运行之前，需要将数据从主机端的内存传送到 GPU 的显存(也就是 GPU 的内存)中，在内存中完成计算之后，再将计算后的结果传送回主机端的内存中。同样，CUDA 的学习重点之一也是了解并学会如何合理地使用 GPU 中的存储器和内存。图 4-67 所示为 CUDA 的寄存器模型。

在使用 CUDA 进行并行计算时，通常涉及的寄存器模型有如下八种：

① Register(寄存器)

② Local Memory(局部寄存器)

③ Shared Memory(共享存储器)

④ Constant Memory(常数存储器)

⑤ Texture Memory(纹理存储器)

⑥ Global Memory(全局存储器)

⑦ Pinned Memory(页锁定存储器)

⑧ Pageable Memory(可分页存储器)

本节将对这八种存储器进行简单的介绍。

首先是最底层的寄存器(Register)。对每个线程来说,寄存器都是私有的,这与CPU中一样。如果寄存器被消耗完,数据将被存储在本地存储器(Local Memory)中。本地存储器对每个线程来说也是私有的,但是此时数据将会被保存在帧缓冲区DRAM中,而不是片内的寄存器或者缓存中。线程的输入和中间输出的变量将被保存在寄存器或者本地存储器中。

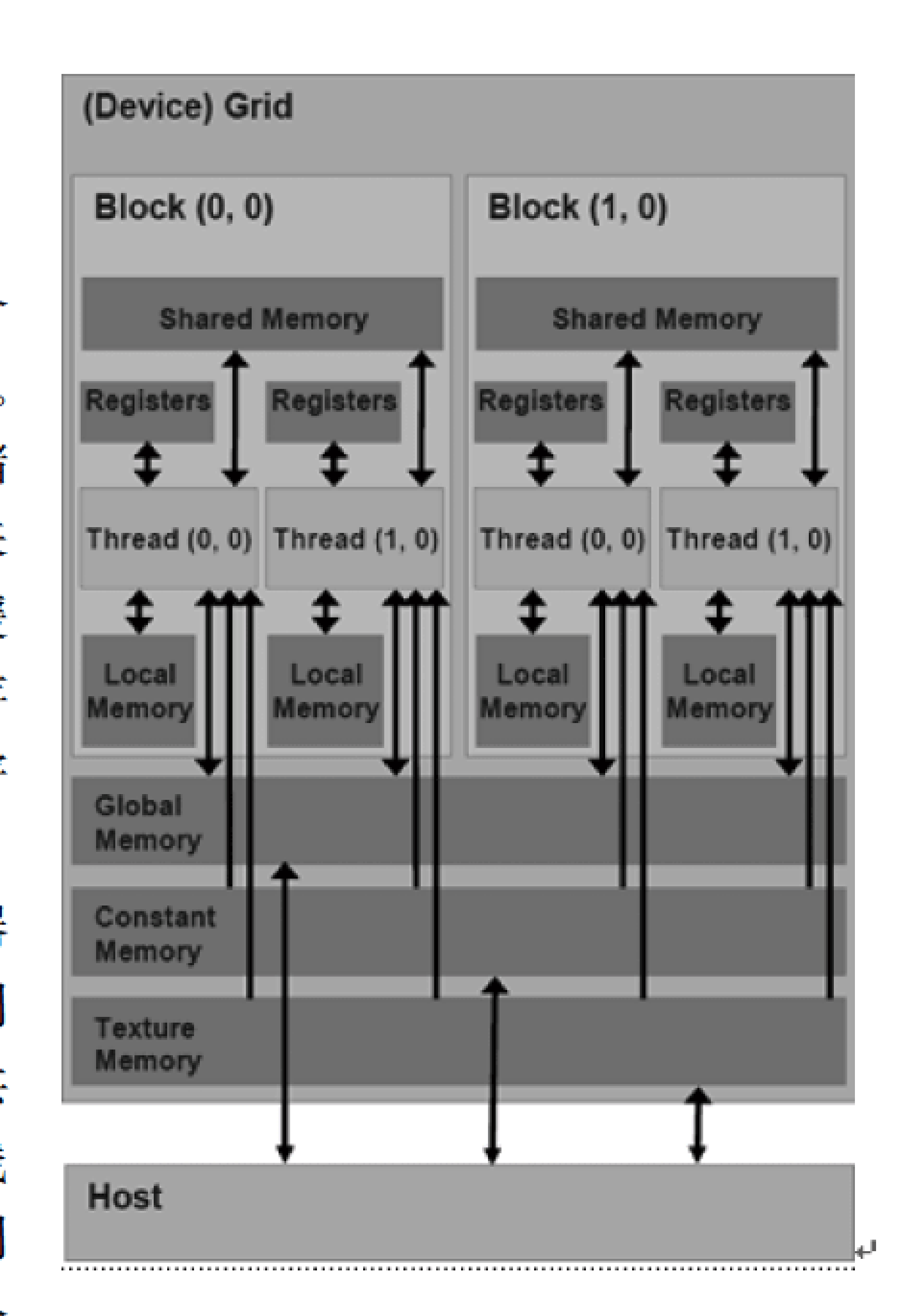


图 4-67　CUDA的存储器模型

之后是用于线程间通信的共享存储器(Shared Memory)。共享存储器是一块可以被同一块中的所有线程访问的可读写存储器。访问共享存储器几乎和访问寄存器一样快,可最小化线程间的通信延迟。共享存储器可以实现许多不同的功能,例如用于保存公用计数器和计算块的公用结果等。

除此以外,还有两种只读的地址空间——常数存储器(Constant Memory)和纹理存储器(Texture Memory),它们是由GPU中用于图形计算的专用单元发展而来的。常数存储器空间较小(通常只有64KB),支持随机访问。纹理存储器容量则大得多,并且支持二维寻址。这两种存储器实际存在于帧缓冲区DRAM中,但由于它们的“只读”性质,在GPU片内可以进行缓存,这样可以加快访问速度。这两种存储器并不要求缓存一致性,但这也意味着如果CPU或者GPU要更改常数存储器或者纹理存储器的值,缓存中的值在更新完成之前将无法使用。

再之后是全局存储器(Global Memory),使用的是普通显存。整个线程网格中的任意线程都能读写全局存储器的任意位置,既可以从主机端进行访问,也可以从设备端进行访问。由于全局存储器是可写的,GPU片内没有对其进行缓存。

最后是主机端的两个寄存器:页锁定存储器(Pinned Memory)和可分页存储器(Pageable Memory)。这两个存储器的调用方式与平时在C语言和C++语言中是一样的,此处不再赘述。

4.8.1　寄存器

寄存器(Register)

位置:GPU片内

是否拥有缓存:N/A

访问权限:设备可进行读/写操作

变量生存周期:与线程相同

寄存器是一种高速缓存器,有着极低的访问延迟和最高的效率,主要用于存储程序中的

局部变量。每个寄存器文件的大小为32bit。不同计算能力的GPU，每个SM上寄存器的个数都不一样，在计算能力1.0和1.1的版本中，每个SM中寄存器文件数量为8192；而在1.2/1.3的版本中，每个SM中寄存器文件数量为16 384。也就是说，如果一个SM被划分成八个块，那么平均每个块可以使用8192÷8=1024个寄存器文件。寄存器动态分配，一旦分配到某一个块，不能被其他块访问。如果每个块包含16×16=256个线程，那么每个线程只能使用四个寄存器文件。

程序编译时，就已确定每个线程可以使用多少寄存器，所有块的寄存器个数相同。需要注意的是，如果每个线程都使用了过多的寄存器，那么将导致每个SM上同时执行块的个数减小。例如，如果每个线程使用八个寄存器文件，那么该SM上最多同时执行四个块(每个块有256个线程)。因此，可以总结为以下两点：

- 少量线程，每个线程使用大量寄存器。
- 大量线程，每个线程使用少量寄存器。

4.8.2 局部存储器

局部寄存器(Local Memory)

位置：板载显存

是否拥有缓存：无

访问权限：设备可进行读/写操作

变量生存周期：与线程相同

局部寄存器是每个CUDA线程私有的，通常情况下，在声明数组或者寄存器被使用完毕、大型结构体或数组、无法确定大小的数组、线程的输入和中间变量或者是定义线程私有数组的同时进行初始化的数组，这些都将被分配到局部寄存器中。

局部寄存器在板载显存中，因此进行访问时会变得很慢。

4.8.3 共享存储器

共享存储器(Shared Memory)

位置：GPU片内

是否拥有缓存：N/A

访问权限：设备可进行读/写操作

变量生存周期：与块相同

在CUDA中，共享存储器是GPU片内除寄存器外的另一种高速可读可写的存储器，同一个线程块内的线程均可以访问它，一般用于存储块中线程公用的数据。它提供了线程间通信的机制，并且访问速度十分接近寄存器的速度。但共享存储器在被访问时可能会产生存储体冲突(bank conflict)，导致访问效率低下，因此，需要对CUDA程序进行优化，以减少冲突的次数。

前面已经介绍了如何将两个向量数组对应位置的数据进行加和，现通过点积运算的方式来介绍如何使用共享内存。

数量积(dot product，scalar product，也称为点积)是接受在实数域R上的两个向量并返回一个实数值标量的二元运算。它是欧几里得空间的标准内积，公式表示如下：

$$\boldsymbol{a} = (a_1, a_2, a_3);$$
$$\boldsymbol{b} = (b_1, b_2, b_3);$$
$$\boldsymbol{a} \cdot \boldsymbol{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

那么用一维数组的形式来表示点积，就是

$$\boldsymbol{a} = [a_1, a_2, a_3];$$
$$\boldsymbol{b} = [b_1, b_2, b_3];$$
$$\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b} = [a_1 * b_1,\ a_2 * b_2, a_3 * b_3];$$

现在如果使用 GPU 将两个长度为 10 000 的数组进行内积，就需要使用共享内存。需要注意一点，在使用共享存储器进行叠加时，GPU 内部的实际叠加流程其实是按如图 4-68 所示的方式进行的，这样可以保证速度的最大化。

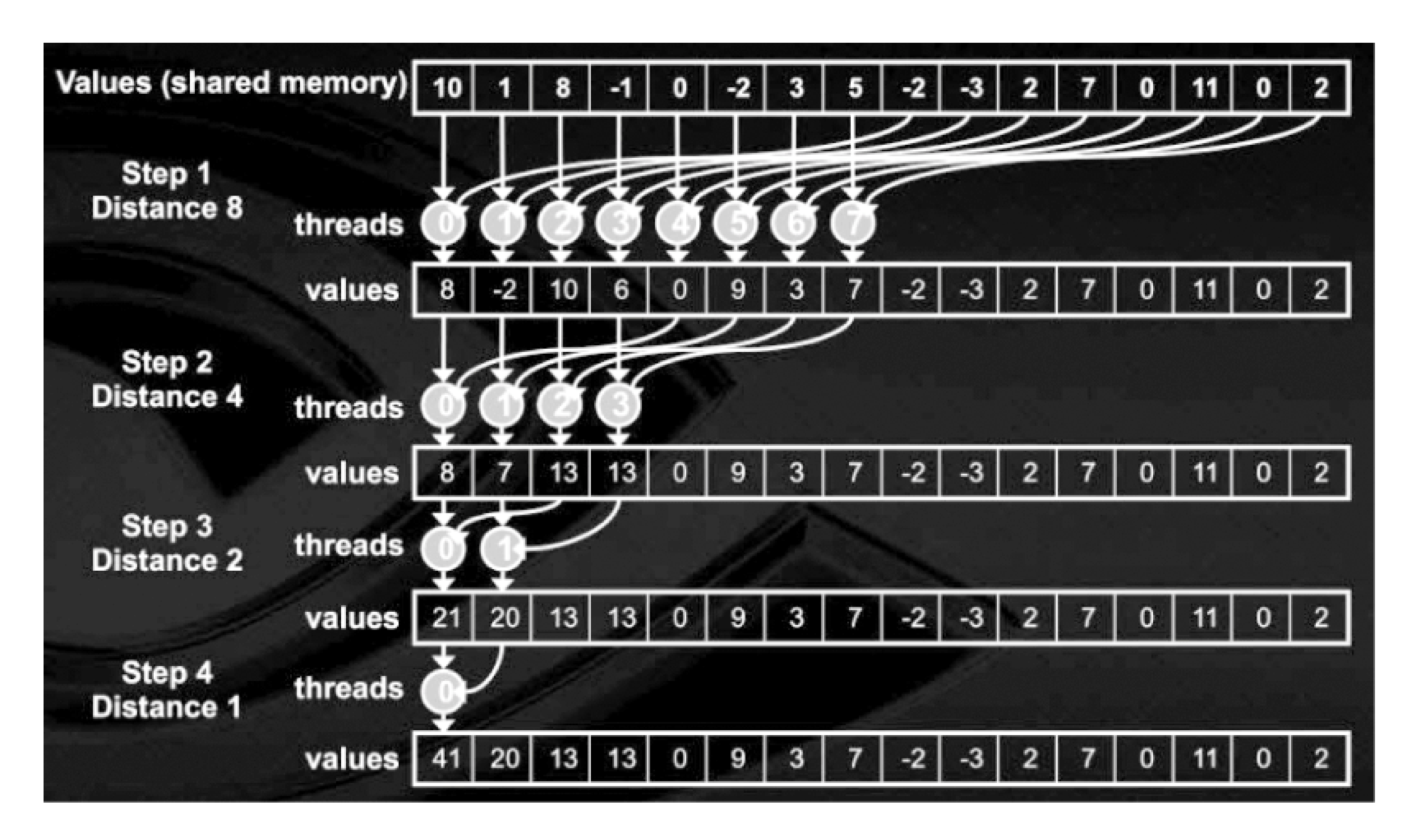


图 4-68　共享存储器应用

使用的程序和程序运行后的结果如下：

```
#include <iostream>
#include <cuda_runtime.h>
#include <device_functions.h>
#include <device_launch_parameters.h>

const int N = 10000;
const int thread = 128;
const int block = 64;

__global__ void scalar( float *a, float *b, float *c ) {
    __shared__ float share[thread];
    int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    int shareIndex = threadIdx.x;

    float temp = 0;
    while (threadID < N) {
        temp += a[threadID] * b[threadID];
        threadID += blockDim.x * gridDim.x;
```

```
        }

        share[shareIndex] = temp;

        //让线程块内的所有线程运行完毕
        __syncthreads();

        //规约运算
        int i = blockDim.x/2;
        while (i != 0) {
            if (shareIndex < i)
                share[shareIndex] += share[shareIndex + i];
            __syncthreads();
            i /= 2;
        }
        if (shareIndex == 0)
            c[blockIdx.x] = share[0];
}

int main( void ) {
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;

    //CPU 分配内存
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( block*sizeof(float) );

    //GPU 分配内存
    cudaMalloc( (void**)&dev_a,N*sizeof(float) );
    cudaMalloc( (void**)&dev_b,N*sizeof(float) );
    cudaMalloc( (void**)&dev_partial_c,block*sizeof(float) );

    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;
    }

    cudaMemcpy( dev_a, a, N*sizeof(float),cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N*sizeof(float),cudaMemcpyHostToDevice );

    scalar<<<block,thread>>>( dev_a, dev_b,dev_partial_c );

    cudaMemcpy( partial_c, dev_partial_c,block*sizeof(float),cudaMemcpyDeviceToHost );

    c = 0;
    for (int i=0; i<block; i++) {
        c += partial_c[i];
    }

    printf( "内积的结果%f\n", c );
```

```
    //释放 GPU 指针
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_partial_c );

    //释放 CPU 指针
    free( a );
    free( b );
    free( partial_c );

    getchar();

}
```

程序运行的结果如图 4-69 所示。

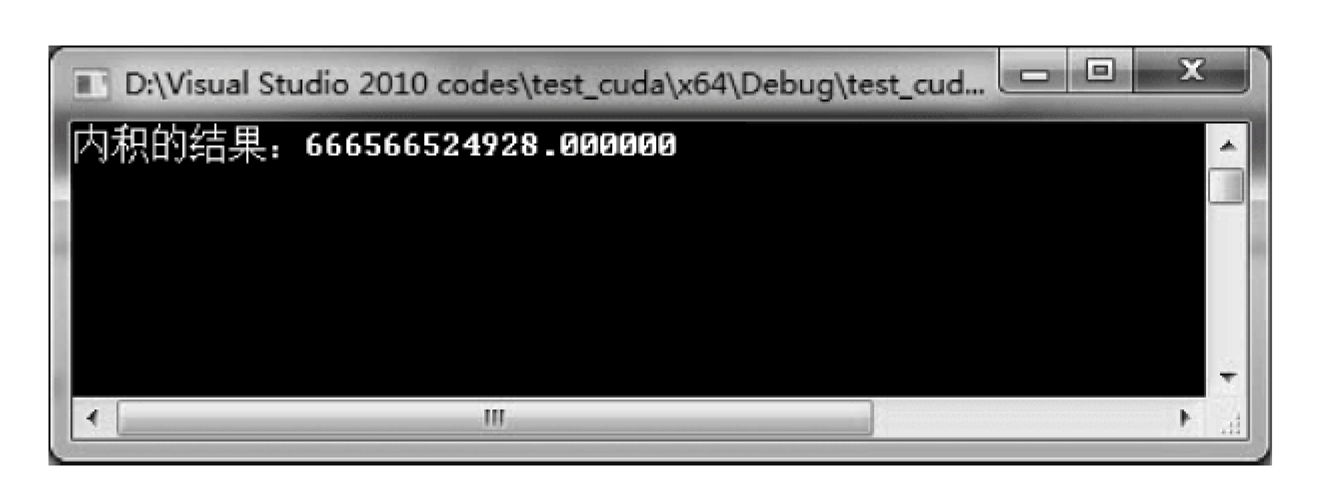


图 4-69　点积的结果

4.8.4　常数存储器

常数存储器(Constant Memory)

位置：板载显存

是否拥有缓存：有

访问权限：设备可进行读操作，主机可进行读/写操作

变量生存周期：可在程序中保持

常数存储器位于 GPU 片外显存，其地址空间是只读的，且使用了缓存加快访问速度。常数存储器只占用了 64KB，一般用于存储程序中需要经常访问的只读参数，并且整个网格共享一个常数存储器。在访问常数存储器时若发生缓存命中，则访问所需数据只需要一个时钟周期的时间。在一些情况下，使用常数存储器而不使用全局存储器，可以削减带宽。

4.8.5　纹理存储器

纹理存储器(Texture Memory)

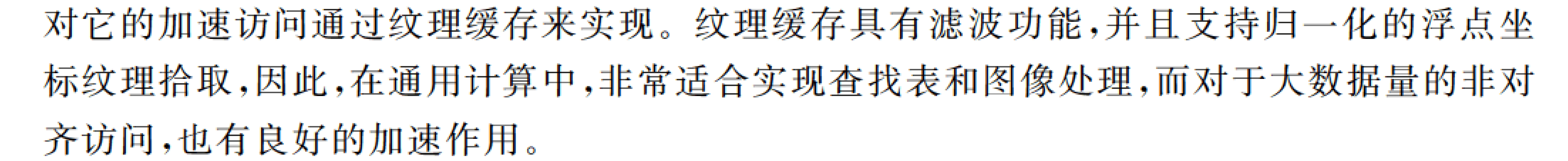
位置：板载显存

是否拥有缓存：有

访问权限：设备可进行读操作，主机可进行读/写操作

变量生存周期：可在程序中保持

纹理存储器是只读的。它的前身是渲染纹理的图形专用单元，故具有一些特殊功能，并且整个线程网格共享一个纹理存储器。纹理存储器支持一维、二维和三维数组的存储形式，



对它的加速访问通过纹理缓存来实现。纹理缓存具有滤波功能，并且支持归一化的浮点坐标纹理拾取，因此，在通用计算中，非常适合实现查找表和图像处理，而对于大数据量的非对齐访问，也有良好的加速作用。

4.8.6 全局存储器

全局存储器(Global Memory)

位置：板载显存

是否拥有缓存：无

访问权限：设备可进行读/写操作

变量生存周期：可在程序中保持

全局存储器位于 GPU 片外的板载显存中，CPU 和 GPU 都能对其进行读写访问，并且整个线程网格中的所有线程也都能随意读写它的任意位置。Fermi 架构之前的 GPU 是没有全局存储器配备缓存的，所以访问速度较慢，但是它能提供很高的带宽，一般用于存储大规模的公用数据。使用全局存储器时，应遵守对齐访问(coalesced access)的要求，才能避免分区冲突，有效利用带宽，否则会导致访问效率低下。

4.8.7 页锁定存储器

页锁定存储器(Pinned Memory)

位置：主机内存

是否拥有缓存：无

访问权限：主机可进行读/写操作

变量生存周期：可在程序中保持

主机端页锁定存储器只分配在主机端的物理内存上，而且地址固定，它能通过 DMA 加速与设备端的通信，可以有效提高主机端与设备端的通信效率，此外，只有主机端页锁定存储器才能使用 CUDA API 提供的异步传输功能，允许在 GPU 进行计算时完成主机和设备间的通信，实现流式处理。尽管页锁定存储器有很多好处，但是也不能在主机端内存中过多地分配，因为这样可能会使操作系统和其他应用程序没有足够的物理内存而不得不使用虚拟内存，进而导致系统整体性能的下降。页锁定存储器通过函数 cudaHostAlloc() 和 cudaFreeHost() 分配和释放。通常情况下，页锁定存储器只能由分配它的 CPU 线程访问，但是在使用时加上 cudaHostAllocPortable 的标志，就可以由控制不同 GPU 的多个 CPU 线程共享。

4.8.8 可分页存储器

可分页存储器(Pageable Memory)

位置：主机内存

是否拥有缓存：无

访问权限：主机可进行读/写操作

变量生存周期：可在程序中保持

可分页存储器的使用和一般的 C 语言程序相同，通过操作系统 API 中的 malloc()、

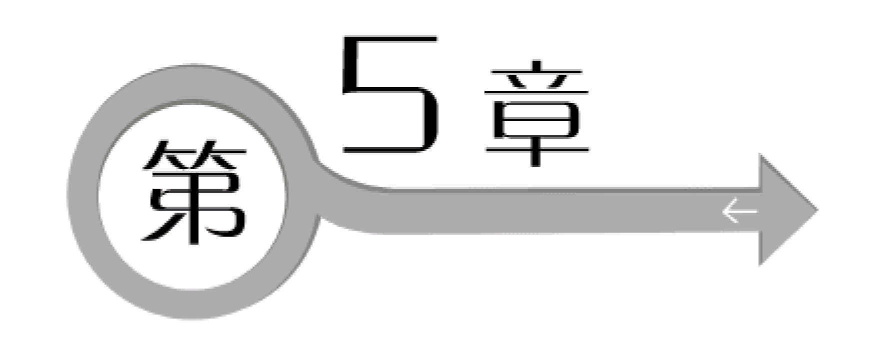
new()、free()分配和回收，但可能会涉及分页内存管理的页面置换算法等，分配的存储空间有可能是低速的虚拟内存。

4.9　本章小结

本章主要介绍了 CUDA 的相关知识。4.1 节主要介绍了 CUDA 的发展历史和应用背景。4.2 节主要介绍了 GPU 的内部结构、具体的架构以及目前市面上常见的 GPU 型号。4.3 节主要介绍了并行处理的基础知识，以及如何使用 GPU 完成并行处理。4.4 节主要介绍了如何安装 CUDA 6.5 并给出一个用于环境测试的小程序。4.5 节主要介绍了 C 语言的最小扩展集以及运行时库。4.6 节介绍了几个简单的并行处理程序，并介绍了一些 CUDA 常用的函数。4.7 节针对 CUDA 最核心的线程，通过理论加示例的方式进行了详细讲解。4.8 节主要介绍了 GPU 内最常用的八种存储器，并在介绍寄存器的过程中给出了示例程序。

参考文献

[1]　丁鹏，贾月乐，张静，等. GPU 结构与通用计算探析[J]. 技术与市场，2009，16(9):4-5.
[2]　刘金硕，邓娟，周峥. 基于 CUDA 的并行程序设计[M]. 北京：科学出版社，2014.



基于GPU的并行图像处理

前面已经介绍了 OpenCV 视觉库的应用背景、环境搭建和应用实例等，但是 OpenCV 本身在开发的过程中是基于 CPU 的，所以 OpenCV 本身不支持并行处理。为了使 OpenCV 可以在并行环境下顺利运行，需要将 OpenCV 源程序嵌入支持并行处理的 CUDA 程序中，并重新编译，这样就可以生成一个全新的支持并行处理的 OpenCV 视觉库。新生成的 OpenCV 库，是根据计算机系统的型号（Windows 7 32 位、Windows 7 64 位、Windows 8 64 位等）、CUDA 的版本号、OpenCV 的版本号和 Visual Studio 的版本号来确定的，也就是说，如果这些型号完全一致，则生成的库可移植，理论上是只要有一项不同就无法移植。比如将台式机（Windows 7 32 位）上生成的并行 OpenCV 库移植到笔记本（Windows 7 64 位）上，在运行 CUDA 的时候就会弹出“无法找到并行入口”等一系列问题。为了避免这一系列不必要的问题，需要手动生成一个专属于自己计算机的并行 OpenCV 库。

本章主要介绍如何重新编译支持 GPU 的 OpenCV 库，如何配置环境并给出了几个并行图像处理的例程。

5.1 CMake 和 TBB 的安装

5.1.1 安装 CMake

CMake 是一个跨平台的安装（编译）工具，可以用简单的语句来描述所有平台的安装（编译过程）[1]。CMake 这个名字是 cross platform make 的缩写。虽然名字中含有 make，但是 CMake 和 UNIX 上常见的 make 功能是不同的，而且更为高级。它能够输出各种各样的 makefile 或者 project 文件，能测试编译器所支持的 C++特性，类似于 UNIX 下的 automake。只是 CMake 的组态档取名为 CmakeLists.txt。CMake 并不直接建构出最终的软件，而是产生标准的建构档（如 UNIX 的 Makefile 或 Windows Visual C++的 projects/workspaces），然后再以一般的建构方式使用。这使得熟悉某个集成开发环境（IDE）的开发者可以用标准的方式建构其软件，这种可以使用各平台原生建构系统的能力是 CMake 与 SCons 等其他类似系统的区别之处。CMake 可以编译程序、制作程序库、产生适配器（wrapper），还可以用任意的顺序建构执行档。CMake 支持 in-place 建构（二进档和源程序

在同一个目录树中)和 out-of-place 建构(二进档在别的目录里),因此可以很容易地从同一个源代码目录树中建构出多个二进档。CMake 也支持静态与动态程序库的建构。

本章使用了 Cmake 编译源代码的功能,将 OpenCV 和 CUDA 放在一起编译生成支持并行处理的 OpenCV 库,选用的 CmMke 型号是 Cmake 3.4.3。

CMake 的下载地址为:

```
https://cmake.org/download/
```

下载 CMake 后就进入安装阶段,因为 CMake 安装较简单,不需要特殊配置,而且各个版本的安装步骤几乎一样,所以本书仅用 Cmake-3.4.3-win32-x86 进行示范,其他版本都可以参考。请注意,这里的 win32-x86 并不仅仅适用于 Windows 32 位系统的计算机,因为它只是用于生成库,与计算机系统无关。

双击安装包,单击"下一步"和"我接受"进入路径选择界面,如图 5-1 所示,选择 Add CMake to the system PATH for all users 单选按钮,让 CMake 自动去系统路径添中加 CMake 路径,选中 Create CMake Desktop Icon 复选框,在桌面创建一个快捷方式,之后选择一个安装地址就可以一步一步安装完成。

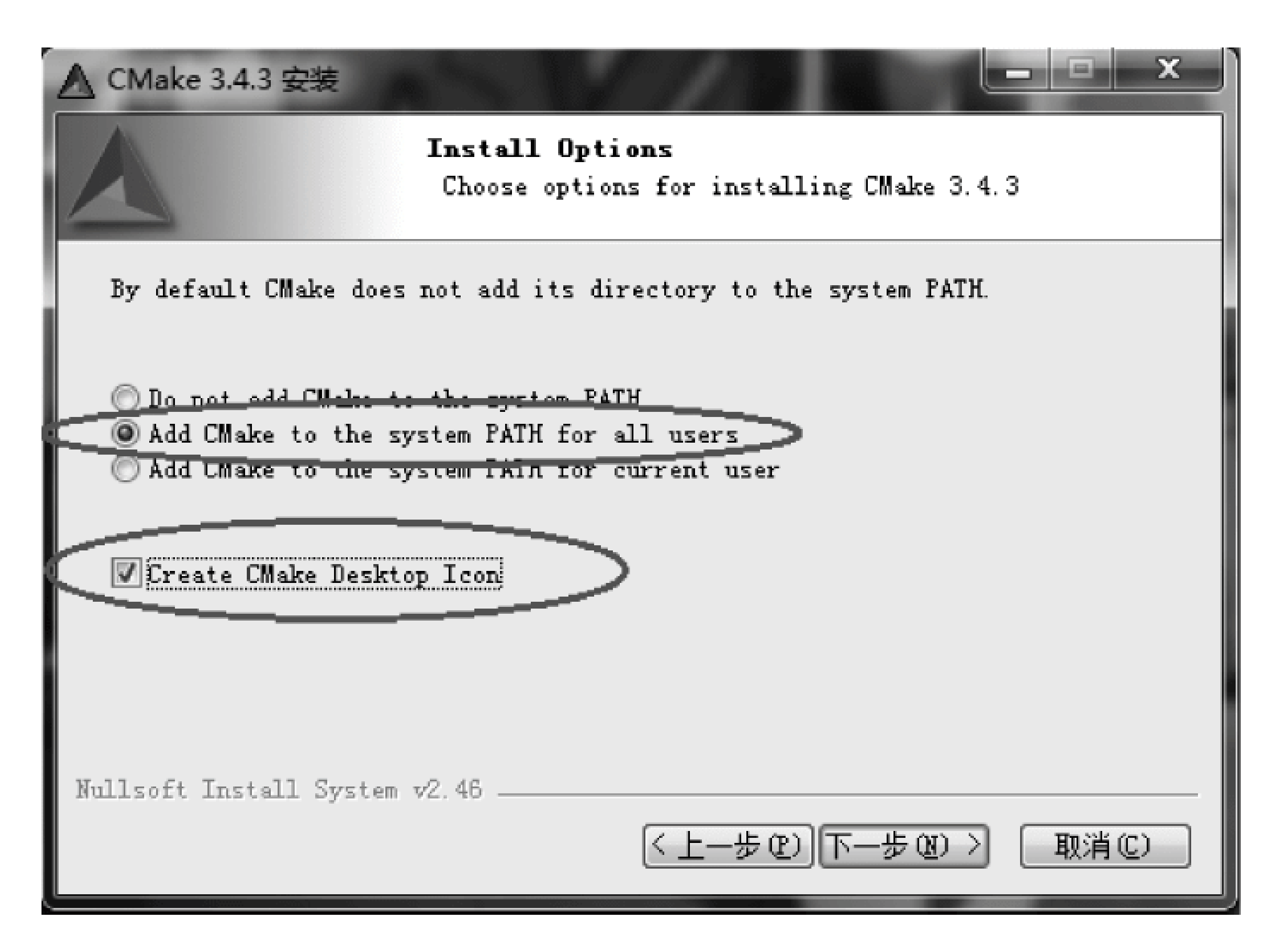


图 5-1　CMake 安装

CMake 3.4.3 在安装完成后会自动在计算机的环境变量中填上自己的路径。在选用 CMake 时尽量选择版本比较高的,高版本不但更方便,而且不会给硬件造成过大的负担,CMake 安装成功会自动运行,如图 5-2 所示。

5.1.2　安装 TBB

TBB 本身就是一个函数库,因此不需要安装,只需要将下载好的文件解压到一个文件夹中,之后回到计算机的"环境变量"中配置环境变量路径,但请注意所有路径不能包含中文。

正常在使用 OpenCV 和 CUDA 的时候可以不添加 TBB 库,但是部分函数在加添 TBB



图 5-2 CMake 安装成功

库时才能正常使用，所以建议在混合编译的过程中将 TBB 库一同编译进去。当然，如果仅仅是使用一些基础函数还是可以不加入的。如下给出了 TBB 库的安装过程。

TBB 的下载地址为：

```
www.threadingbuildingblocks.org/download
```

下载后解压即可，配置方法如下：

首先在用户变量 PATH 中添加一个 TBB 路径，这个 TBB 路径包含 TBB 内的 vc10 文件夹和 include 文件夹，如图 5-3 所示。vc10 代表使用 Visual Studio 2010，如果使用 Visual Studio 2012，则选择 vc11 文件夹。具体的路径根据安放文件夹的位置来确定，切记不能有中文路径。示例中该文件夹放在 D 盘里，路径为：

```
D:\tbb43_20150424oss\include;
D:\tbb43_20150424oss\bin\intel64\vc10
```

配置完用户变量后继续配置系统变量，如图 5-4 所示，在“系统变量”中添加一项新的变量，取名为 TBB，在路径一栏将之前的两条路径写进去。

至此，便完成了 TBB 的安装和配置。

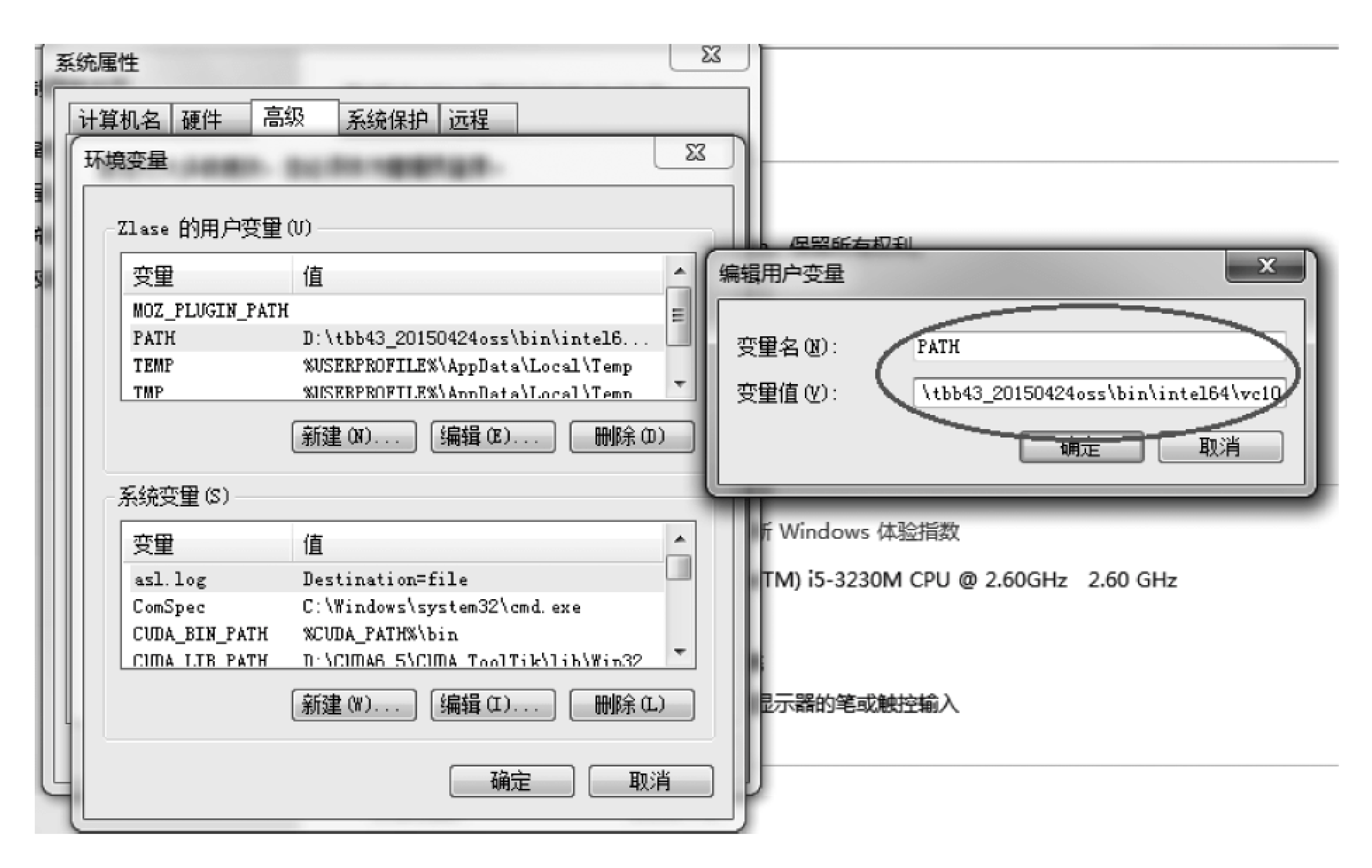


图 5-3　TBB 配置环境第一步



图 5-4　TBB 配置环境第二步

5.2　并行 OpenCV 库的生成

在生成并行 OpenCV 库之前，需要保证计算机中已经成功安装了 VS 2010、TBB 库、CUDA、OpenCV 和 CMake，还要保证这几款软件都配置成功并且能顺利运行。之后生成的过程中，需要确保计算机是在联网的状态下，因为第一次使用 CMake 时需要下载一些

CMake 工作所需的必要文件。

(1) 打开 CMake,如图 5-5 所示,在 Where is the source code 中添加 OpenCV source 的程序文件,示例的路径是:

```
D:/OpenCV2.4.9/opencv/sources
```

其下方的 Where to build the binaries 中添加的是生成文件的存放位置,建议放在一个单独的文件夹中,不要用其默认的路径,示例中存放在了计算机 D 盘的一个名为 test 的文件夹中。

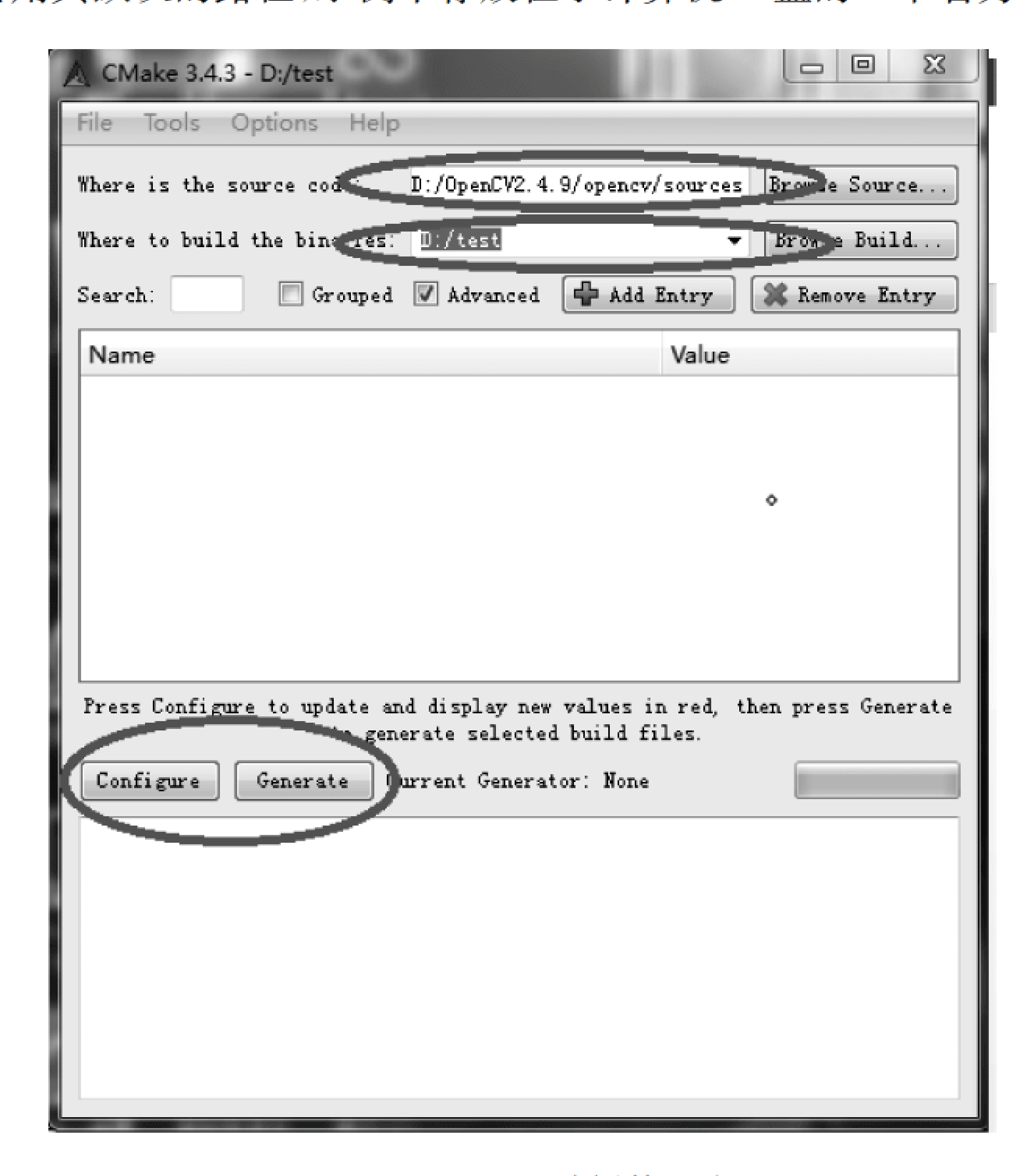


图 5-5 CMake 编译第一步

(2) 单击 Configure 按钮进入选择生成类型的界面,在下拉列表框中选择生成的型号,需要根据计算机的 Visual Studio 版本和 Windows 版本进行选择,示例是 64 位的,因此选择 Visual Studio 10 2010 Win64;如果是 32 位的系统,就选择后边没有 Win64 的选项。

在下边的选项组中可以直接使用默认的 Use default native compilers,如图 5-6 所示,单击 Finish 按钮进入 Configure 阶段。

(3) 单击 Configure 按钮之后需要在列表框中寻找几个必要项,首先选中 WITH_TBB,不要选 BUILD_TBB,为了确保 TBB 可以正常使用,再次单击 Configure 按钮,如果顺利通过最好,如果继续出现关于 TBB 的错误提示,则再次 Configure,这个时候会出现如图 5-7 所示的几行红字。

在这几行错误信息中需要添加几条 TBB 的路径,示例的路径如下:

```
TBB_INCLUDE_DIRS:     D:/tbb43_20150424oss/include
TBB_LIB_DIR:          D:/tbb43_20150424oss/lib/intel64/vc10
TBB_STDDEF_PATH:      D:/tbb43_20150424oss/lib/intel64/vc10
```

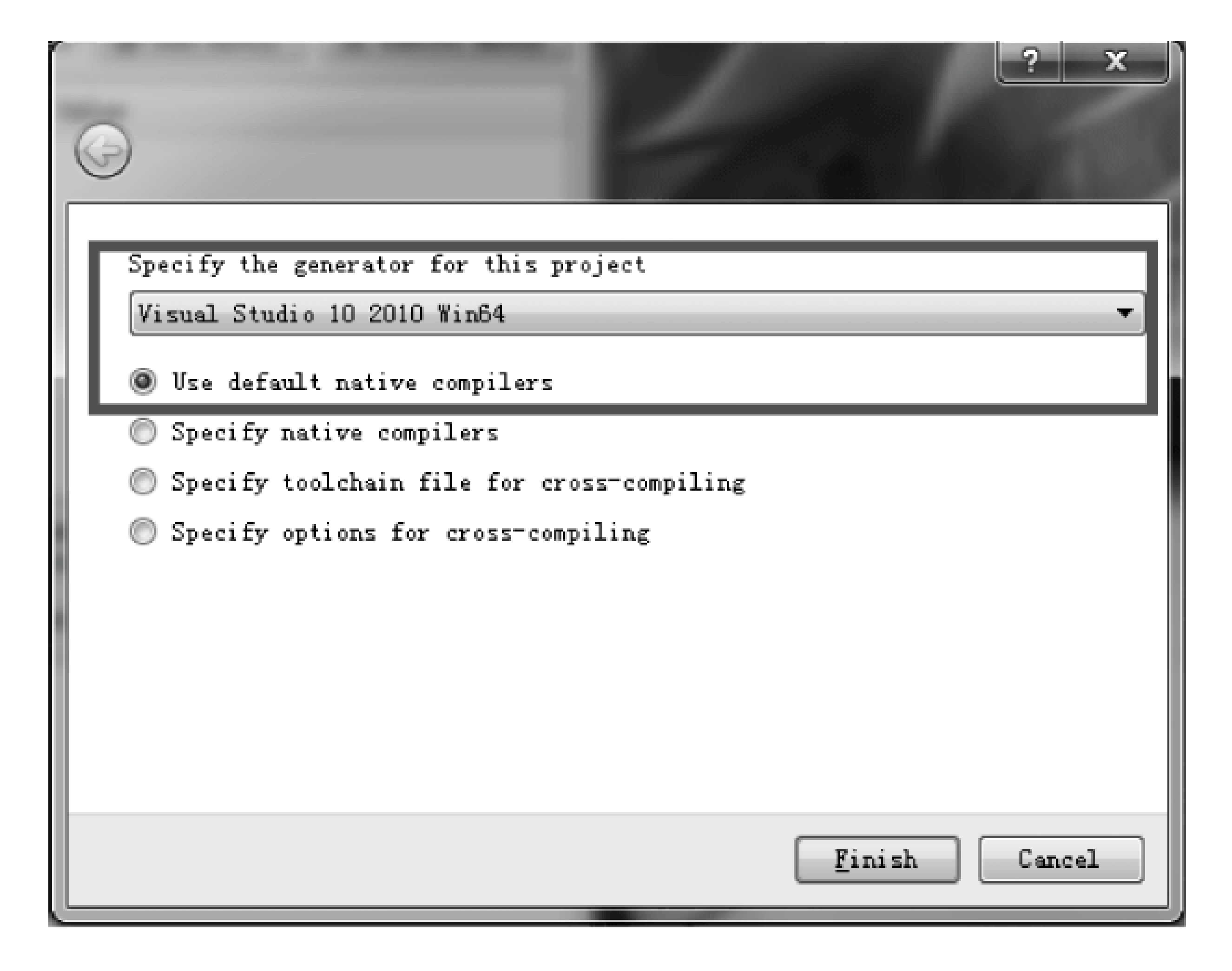


图 5-6 CMake 编译第二步

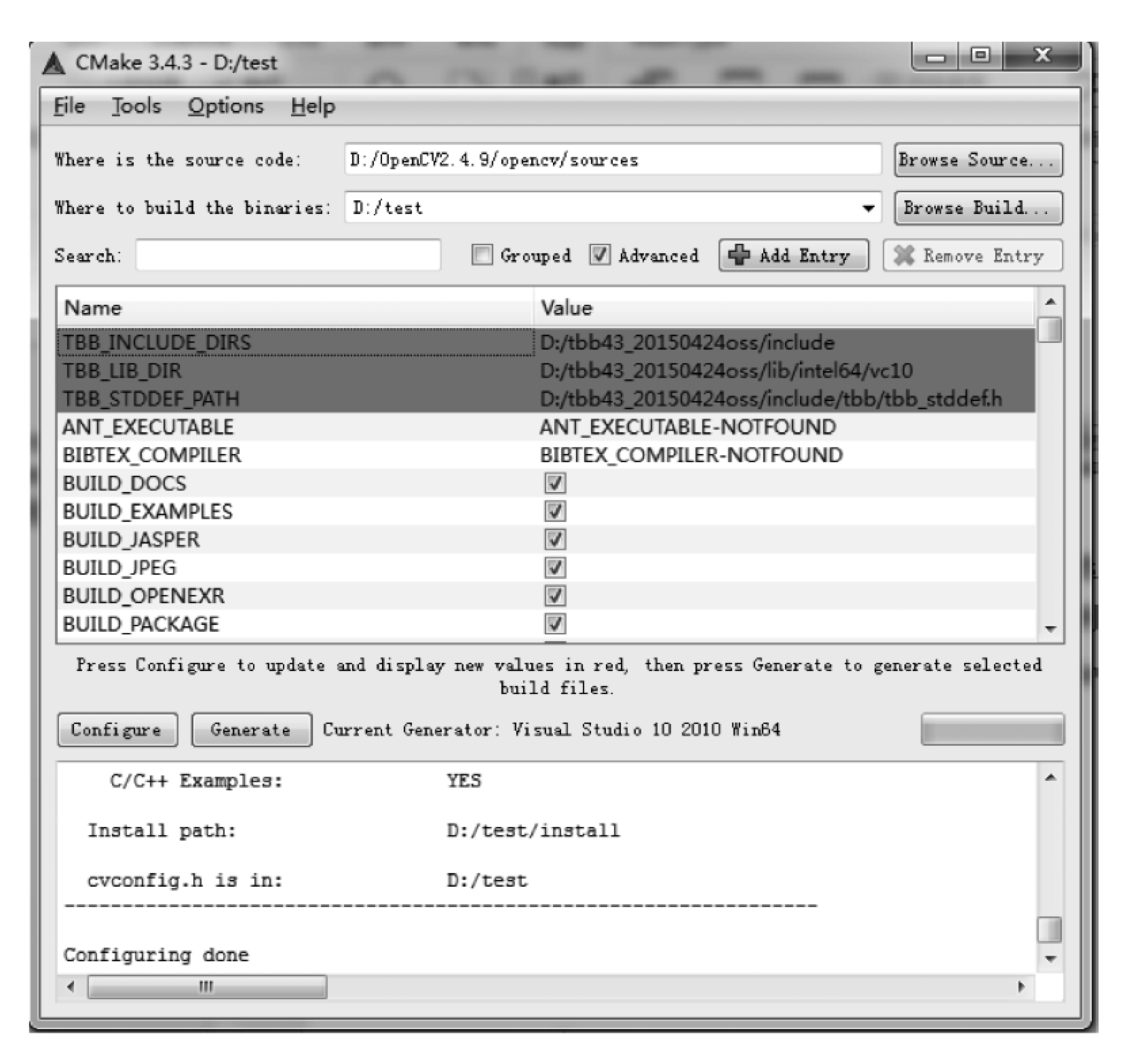


图 5-7 CMake 编译第三步

单击 Configure 按钮多次运行,直到红字消失为止。

补充一点,在选择编译时建议选中 BUILD_EXAMPLE 一项,该选项是询问是否编译例程,虽然编译的过程中例子会占用很多时间,但这对后续的学习有很大的帮助。

(4) 当图 5-7 中的红字都消失后,在列表框中显示的部分找到 Other third-party libraries 选项,注意观察里边的 Use TBB 和 Use CUDA 是否正确,如图 5-8 所示。如果没有这两项,

则说明前面的安装过程有遗漏或错误,也可能是之前安装好的软件没有做测试,需要重新安装出错的软件。

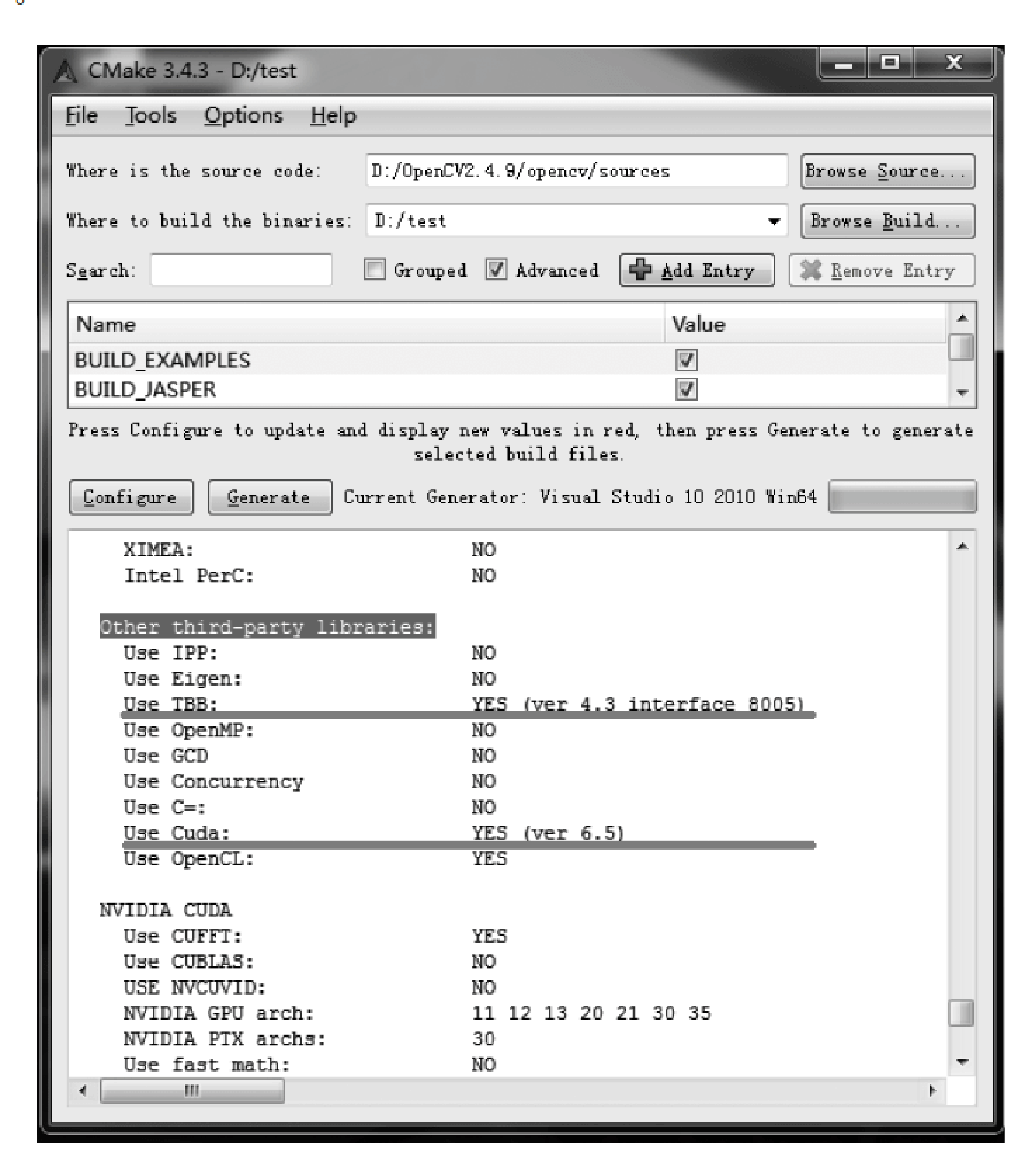


图 5-8 CMake 编译第四步

(5) 确定包含 TBB 和 CUDA 之后单击 Generate 按钮进行生成,生成后即可去之前设定的文件夹中寻找生成的 OpenCV 库,用来进行下一步的编译。生成后的文件夹如图 5-9 所示,找到其中的.sln 文件,用 Visual Studio 2010 打开。

(6) 进入 Visual Studio 2010 中进行编译,如图 5-10 所示。如果在 CMake 中选择生成 EXAMPLE,则对应的解决方案中约有 263 个项目文件;如果选择不生成 EXAMPLE,则对应的解决方案中约有 67 个项目文件。接下来在 Debug 界面中选择平台,如果之前编译的是 64 位就选择 x64,如果是 32 位就选择 Win32,在这些准备工作完成之后右击 ALL_BUILD 进行生成。生成过程耗时较长,单次生成约四小时。Debug 完成之后,将 Debug 改成 Release 重新进行编译,Release 过程也需要消耗同样的时间,整个过程约 7 小时,期间请耐心等待。

(7) Debug 和 Release 都完成后,这个库就成功生成和编译完成了。之后右击 INSTALL,单击"生成"命令,即可将编译好的文件放在 INSTALL 文件夹中,如图 5-11 所示,这个过程 20 分钟左右即可完成。

(8) 回到之前有.sln 的文件夹中,可以找到 install 文件夹,将这个文件夹单独复制取出,该文件夹中装的就是新生成的 OpenCV 文件。

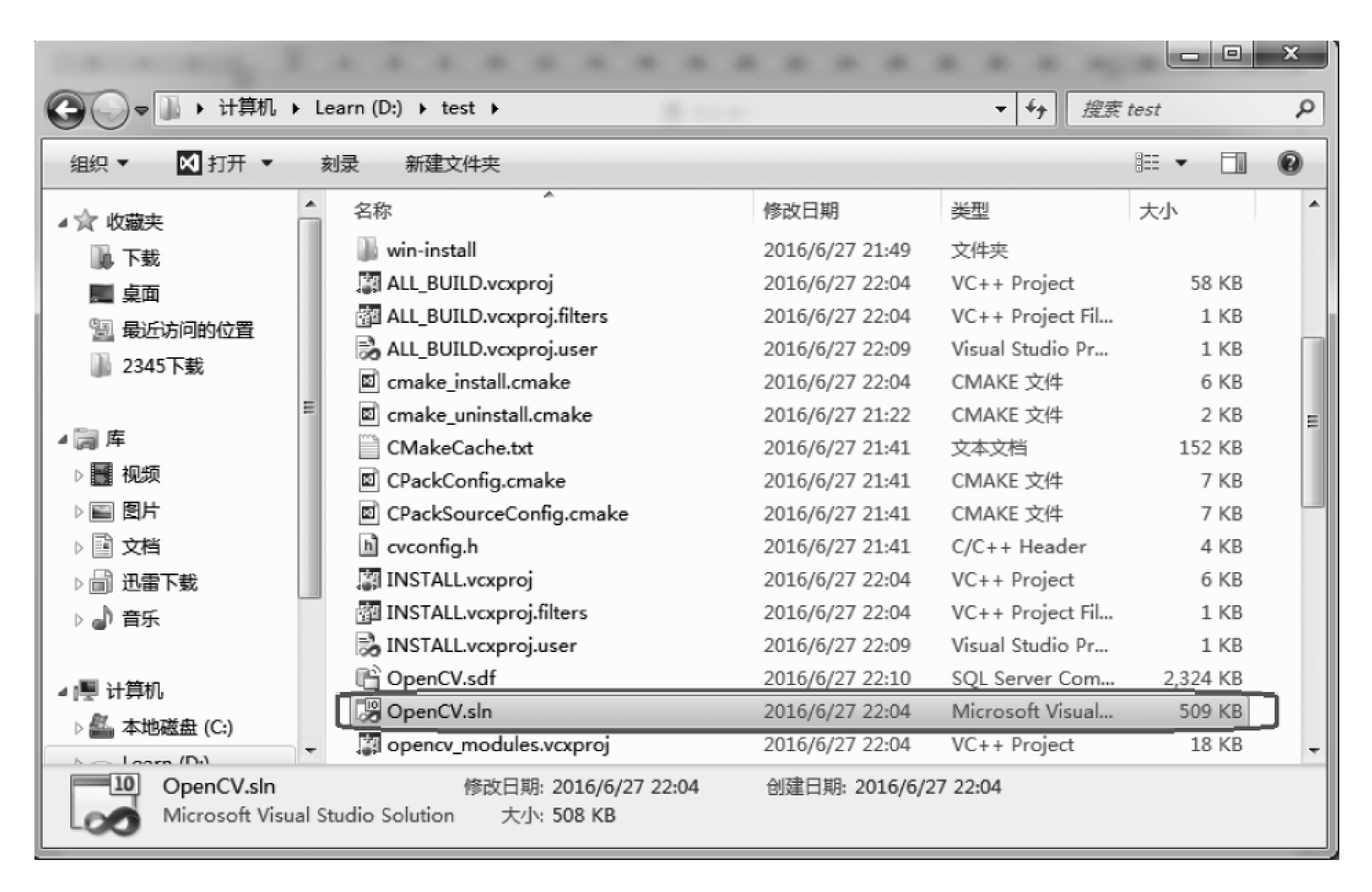


图 5-9　CMake 编译第五步

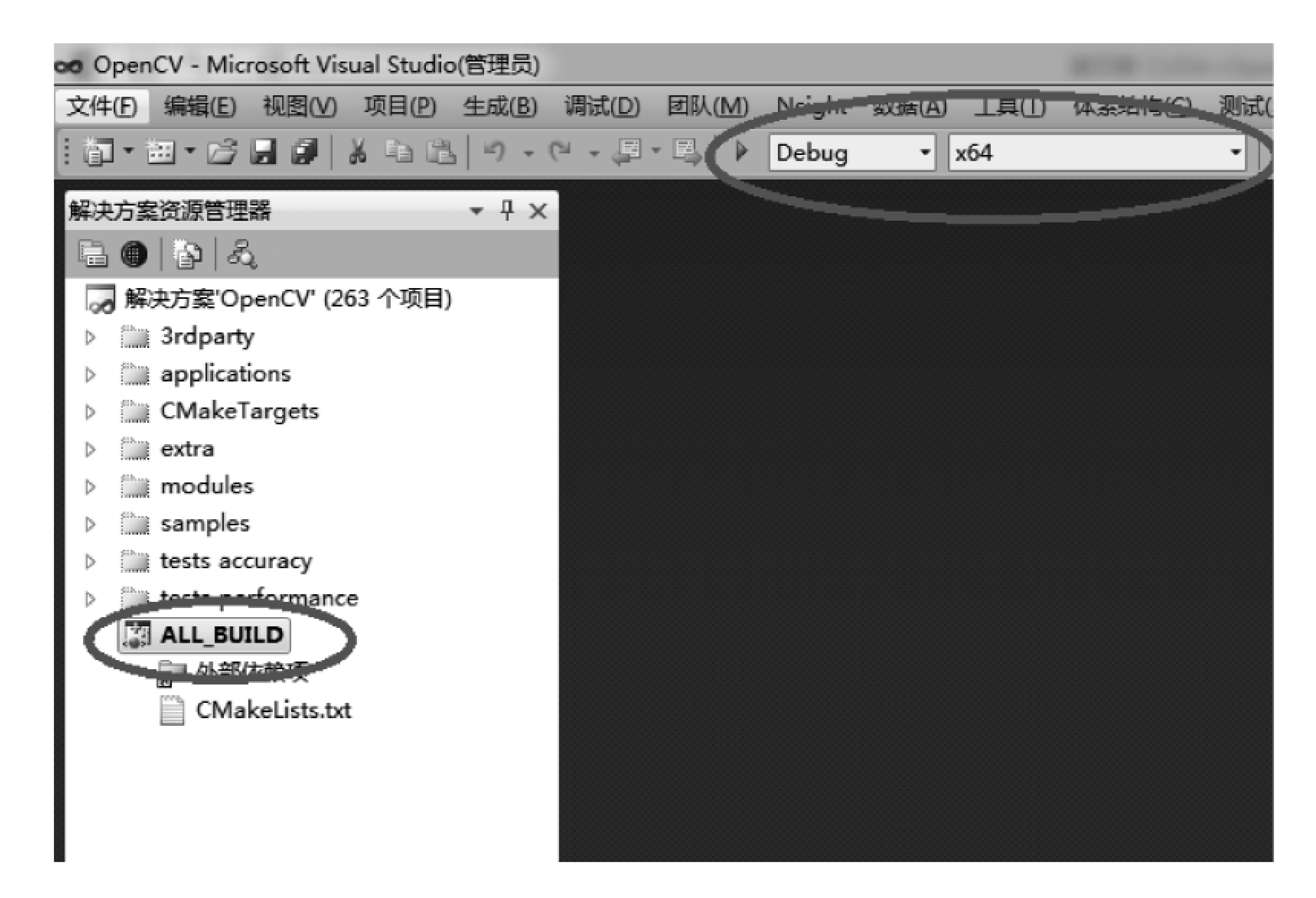


图 5-10　VS 编译第一步

使用新 OpenCV 库前仍然需要配置。新生成的并行 OpenCV 库在不使用 CUDA 时跟原始的 OpenCV 功能一样，所以新生成的 OpenCV 库完全可以替代之前的 OpenCV 库。新 OpenCV 库的配置过程仍然按照第 3 章介绍的配置方法重新配置一遍即可，但是要注意，在配置新 OpenCV 的过程中要用新的路径去替换掉原本的路径，不要让两个版本共存。在编译的过程中路径填错，运行时就会出现错误，并且很难排查，因此必须把之前的 OpenCV 路径全部删除。

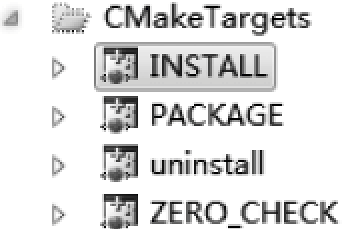


图 5-11　VS 编译第二步

5.3 VS 内的 OpenCV 环境搭建及环境测试

如果想要使用 OpenCV+CUDA 来编程序，可以使用两种方式来配置项目文件。

第一种方式是在原 CUDA 的.cu 文件基础上加上了一个 OpenCV 库，程序开发与前面介绍的在.cu 文件中开发一样，是将 CPU 的程序和 GPU 的程序都写在同一个文件中。第二种方式是在第一种方式的基础上，加上了几个.cpp 文件，用来放置 CPU 中的程序。第二种方式多用在比较大型的项目文件中，大型项目文件如果仅有一个.cu 用来编写程序，会显得很混乱，通过这种方式可以使程序变得更为清晰。

本节将介绍两种环境搭建方式，学习过程中比较推荐第一种配置方式，后面的示例也都是基于第一种配置方式实现的。

5.3.1 常用工程文件的配置

1. 配置工程文件

(1) 这种配置方式和前面的 CUDA 配置很相似，首先需要新建立一个项目文件，如图 5-12 所示。



图 5-12　建立空项目文件

(2) 建立好空项目文件后需要调整其工作平台，因为前面生成的是 64 位的 OpenCV 库，因此需要使用 64 位的平台。如图 5-13 所示。



图 5-13　调整平台

(3) 在“源文件”中添加一个新建项，也就是.cu文件，如图5-14和图5-15所示。



图5-14　添加.cu文件1

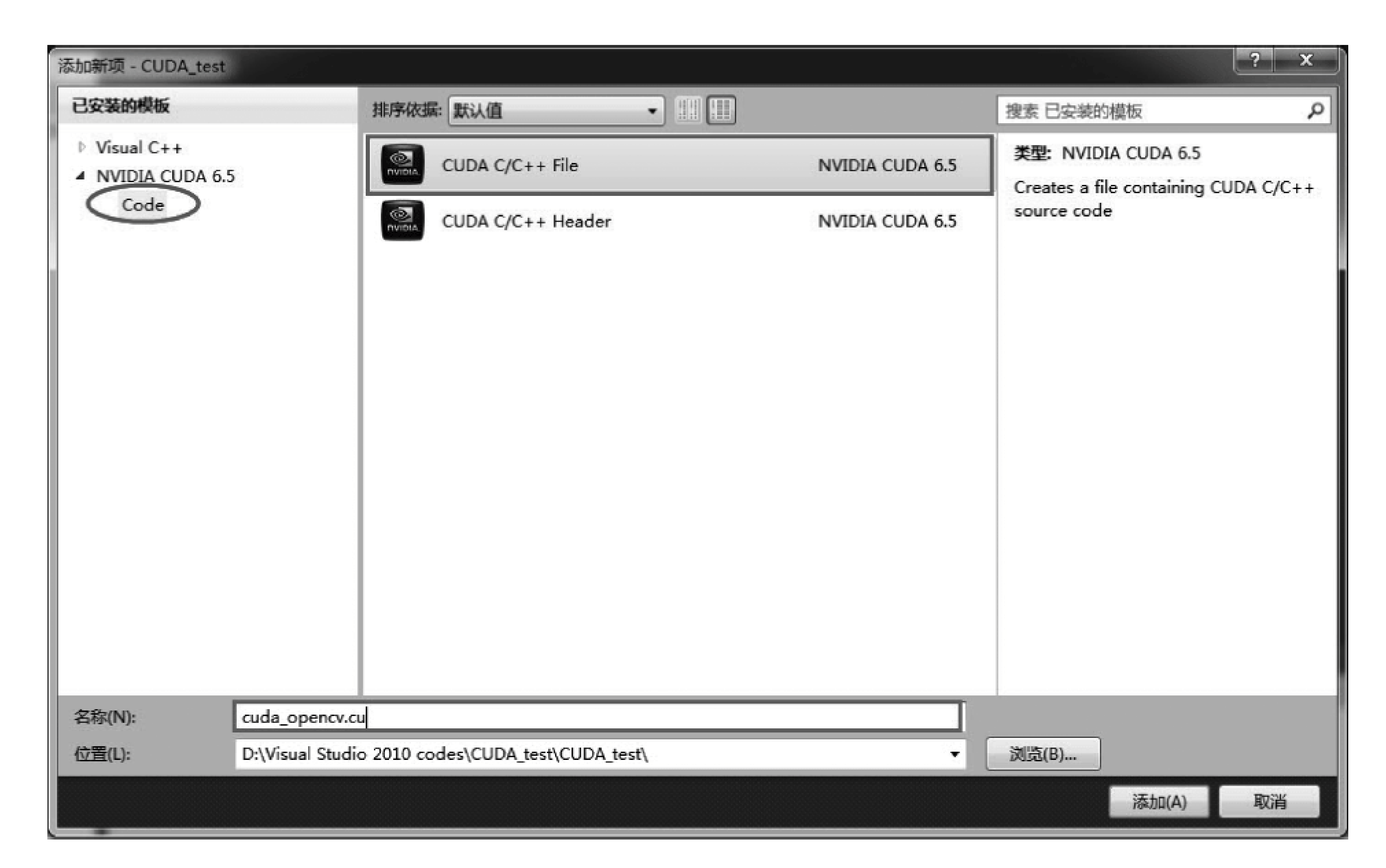


图5-15　添加.cu文件2

(4) 进行工程配置。右击工程文件，选择“属性”命令。在属性界面里找到“VC++目录”选项，在右侧的“包含目录”和“库目录”中依次添加OpenCV和CUDA的头文件以及对应的lib文件。

示例包含目录的路径为：

```
D:\OpenCV2.4.9\install_64\include\opencv2
D:\OpenCV2.4.9\install_64\include\opencv
D:\OpenCV2.4.9\install_64\include
D:\CUDA6.5\CUDA_ToolTik\include
```

库目录为：

```
D:\OpenCV2.4.9\install_64\x64\vc10\lib
D:\CUDA6.5\CUDA_ToolTik\lib\Win32
```

添加完成后的结果如图 5-16 所示。



图 5-16 添加包含目录和库目录

（5）进入附加依赖项的配置，也就是一些对应的 lib 文件。找到“链接器”并在其中找到“输入”选项，在右侧的“附加依赖项”中添加对应的 OpenCV 和 CUDA 的 lib 文件。需要注意的是，在 Debug 和 Release 中添加的 lib 文件是不同的。

示例的包含目录为：

```
cudart.lib
opencv_calib3d249d.lib
opencv_contrib249d.lib
opencv_core249d.lib
opencv_features2d249d.lib
opencv_flann249d.lib
opencv_gpu249d.lib
opencv_highgui249d.lib
opencv_imgproc249d.lib
opencv_legacy249d.lib
opencv_ml249d.lib
opencv_objdetect249d.lib
opencv_ts249d.lib
opencv_video249d.lib
```

第一个是 CUDA 的 lib 文件，其余都是 OpenCV 的 lib 文件，如图 5-17 所示。

（6）打开“清单工具”，将“输入和输出”中的“嵌入清单”这一项改成“否”，如图 5-18 所示。

（7）至此，Bebug 下的配置就完成了，之后单击“应用”按钮，再进入 Release 界面下进行配置，如图 5-19 所示。

Release 与 Debug 不同之处在于“输入”中的附加依赖项，要添加另一组 lib 文件，如图 5-20 所示。



图 5-17　附加依赖项的配置

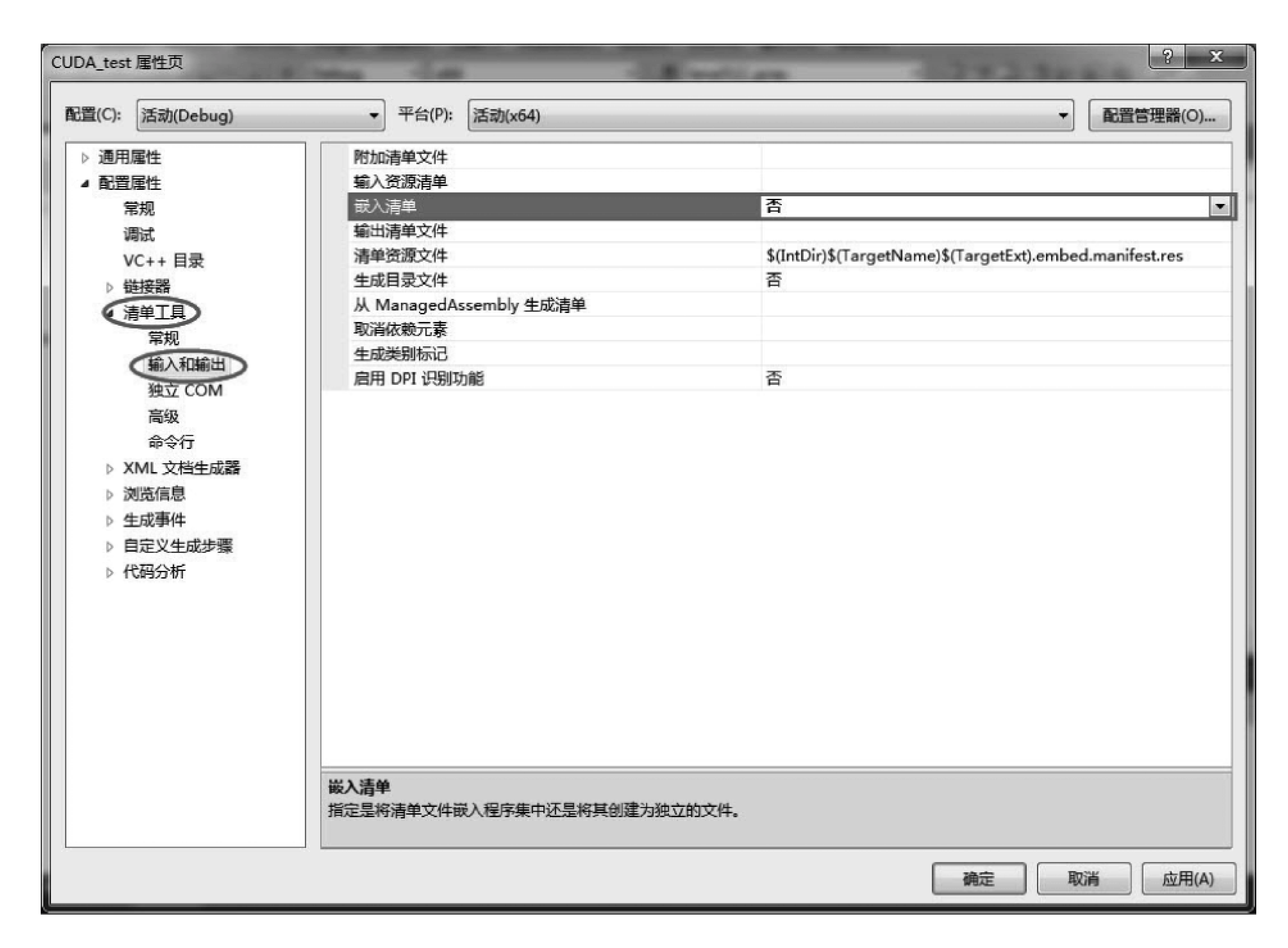


图 5-18　修改嵌入清单



图 5-19 配置 Release



图 5-20 Release 下的附加依赖项

```
cudart.lib
opencv_calib3d249.lib
opencv_contrib249.lib
opencv_core249.lib
opencv_features2d249.lib
opencv_flann249.lib
opencv_gpu249.lib
opencv_highgui249.lib
opencv_imgproc249.lib
opencv_legacy249.lib
opencv_ml249.lib
opencv_objdetect249.lib
opencv_ts249.lib
opencv_video249.lib
```

(8) 回到VS的主界面,右击项目文件,选择"生成自定义"命令,如图5-21所示。



图5-21　项目生成自定义

找到基于CUDA生成,因为版本不同,所以基于项也不完全相同,示例中的生成自定义项文件为CUDA 6.5,如图5-22所示。

(9) 右击.cu文件,选择"属性"命令,如图5-23所示。

在属性界面右侧的"项类型"中选择CUDA C/C++选项,单击"确定"按钮,如图5-24所示,至此,工程文件配置完成。

2. 配置文件测试

完成工程文件配置后,可以写一段程序来进行测试。这里使用了一个图像遍历的程序来作为测试程序,使用CPU读入图像,之后在GPU中将整个图像复制到另一个Mat类变量中,最后再返还给CPU进行输出。

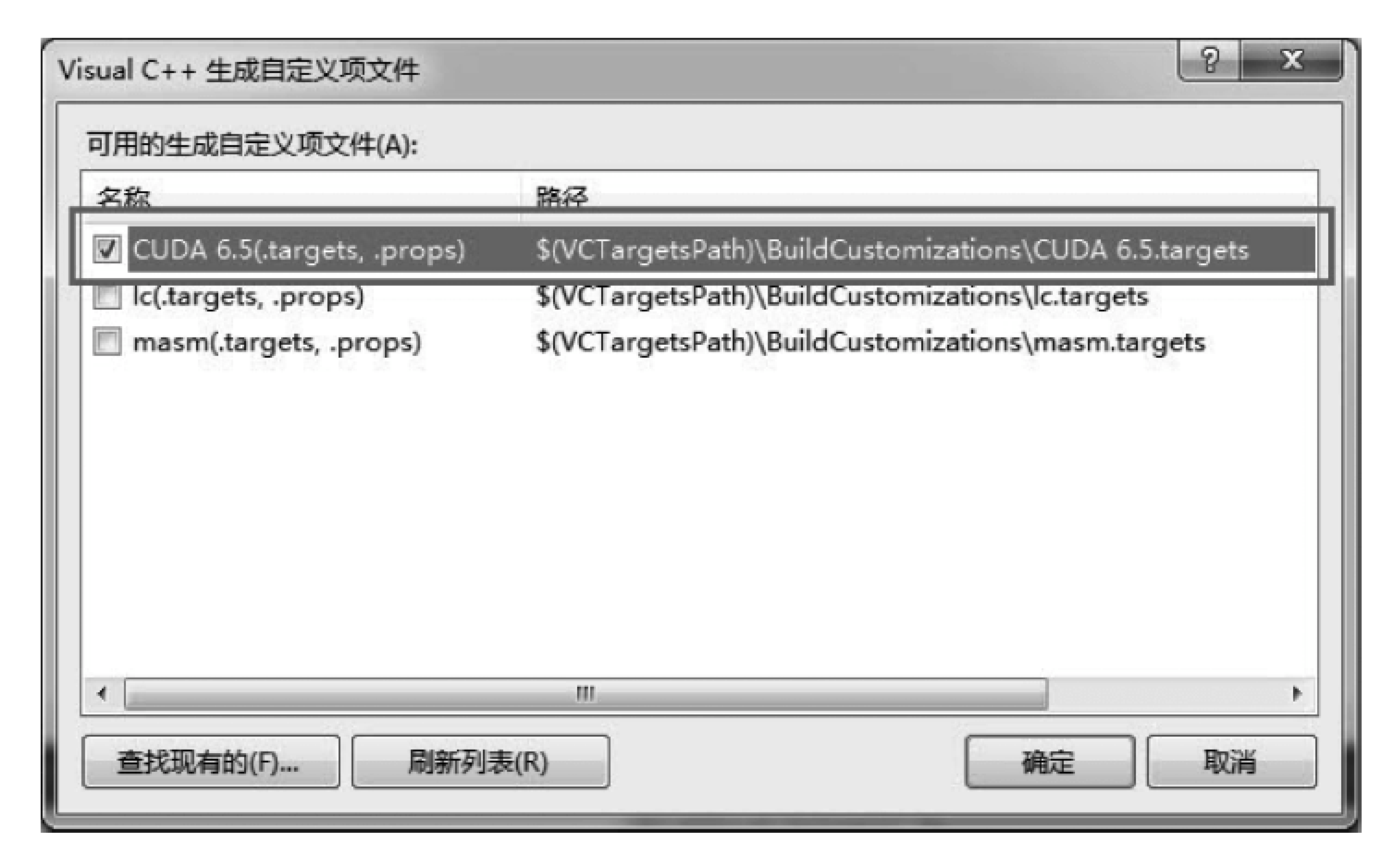


图 5-22　自定义生成



图 5-23　配置.cu 属性



图 5-24　CUDA C/C++生成

程序如下：

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_device_runtime_api.h>
#include <opencv2\gpu\gpu.hpp>
#include <opencv2\gpu\gpumat.hpp>
#include <opencv2\opencv.hpp>
#include <opencv.hpp>
#include <stdio.h>
#include <iostream>
#include "opencv2/gpu/device/common.hpp"
#include "opencv2/gpu/device/reduce.hpp"
#include "opencv2/gpu/device/functional.hpp"
#include "opencv2/gpu/device/warp_shuffle.hpp"
using namespace std;
using namespace cv;
using namespace gpu;

template <int nthreads>
        __global__ void compute_kernel(int height, int width, const PtrStepb img,PtrStepb dst)
        {
            const int x = blockIdx.x * blockDim.x + threadIdx.x;          //x 检索
            const int y = blockIdx.y * blockDim.y + threadIdx.y;          //y 检索

            const uchar * src_y = (const uchar *)(img + y * img.step);
            uchar * dst_y = (uchar *)(dst + y * dst.step);
            if (x < width && y < height)
            {
                dst_y[3 * x] = src_y[3 * x];                              //三通道图像
                dst_y[3 * x + 1] = src_y[3 * x + 1];
                dst_y[3 * x + 2] = src_y[3 * x + 2];
            }
        }


    int main()
  {
      Mat a = imread("lena512.jpg");
      GpuMat d_a(a);
      GpuMat d_dst(d_a.size(),CV_8UC3);

      int width = a.size().width;                                         //图像横向
      int height = a.size().height;                                       //图像纵向
      const int nthreads = 256;
      dim3 bdim(nthreads, 1);
      dim3 gdim(divUp(width, bdim.x), divUp(height, bdim.y));
```

```
    compute_kernel<nthreads><<<gdim, bdim>>>(height,width,d_a,d_dst); //传递到 GPU
    Mat dst(d_dst);
    imshow("原始图像",a);
    imshow("处理后图像",dst);
    waitKey();
    return 0;
}
```

这套程序包含了很多没有用到的头文件，是为了测试配置是否成功，程序运行结果如图 5-25所示。



图 5-25　图像遍历结果

5.3.2　分别配置项目文件

1. 工程配置

这种配置方式是将所有需要运行的程序分成几个部分，就如 CPU 端的程序会放在多个.cpp 文件中，GPU 端运行的程序放在几个.cu 和.cuh 文件中，这种配置可以让程序更清晰明了。配置的方式和使用教程如下：

(1) 在 VS 中新建一个空项目文件，在其中添加一个.cpp 文件，然后在源文件位置处再次添加两个 CUDA 文件，如图 5-26 所示。

在添加新选项界面中选择 NVIDIA CUDA 6.5 下边的 Code 选项，两个文件都选择 CUDA C/C++File 选项。一个文件要以.cu 做结尾，另一个要以.cuh 做结尾，两者没有本质的区别，只是一般用.cu 文件存放 GPU 内要运行的程序，而在.cuh 中要放进 CUDA 的头文件，如图 5-27 所示。

(2) 根据生成的 OpenCV，选择 64 位或 32 位平台，这里都是在 x64 平台下，如图 5-28 所示。

(3) 右击项目选项，单击“属性”命令进行配置。这与之前有些不同，要将“包含目录”“引用目录”和“库目录”三项都要填上，如图 5-29 所示。



图 5-26　建立多个文件

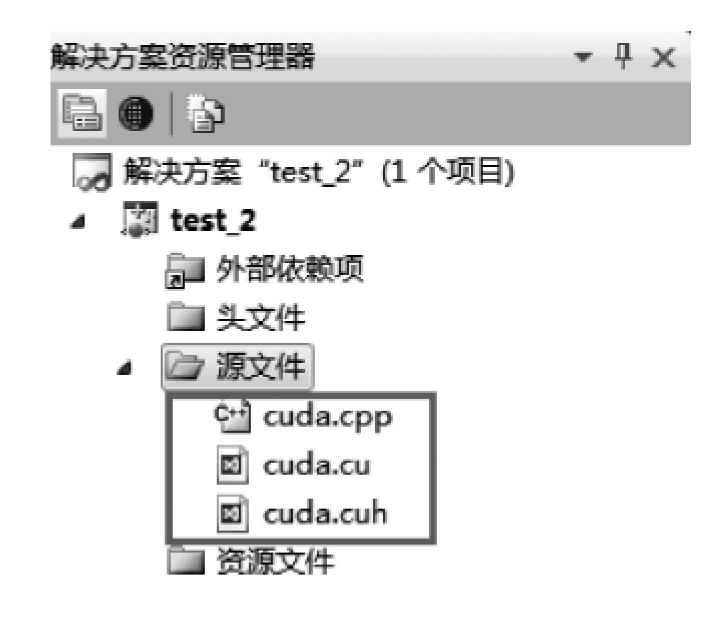


图 5-27　建立三个文件



图 5-28　更换平台

“包含目录”中要填的是 OpenCV 和 CUDA 的目录文件，示例中的路径为：

```
D:\OpenCV2.4.9\install_64\include\opencv2
D:\OpenCV2.4.9\install_64\include\opencv
D:\OpenCV2.4.9\install_64\include
D:\CUDA6.5\CUDA_ToolTik\include
```

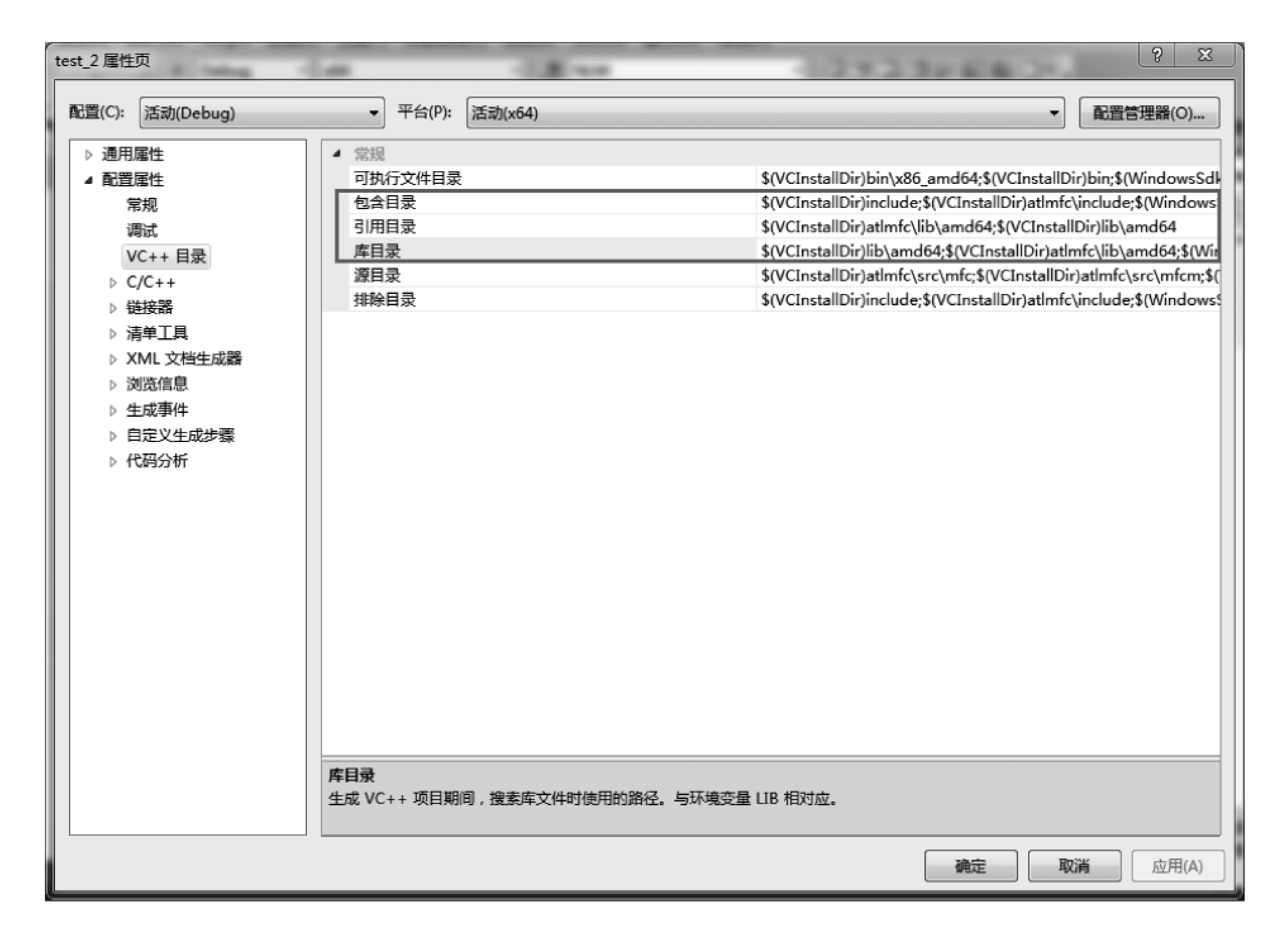


图 5-29　添加包含目录、引用目录和库目录

“引用目录”中要添加 CUDA 中的 lib 项，示例的引用目录为：

```
D:\CUDA6.5\CUDA_ToolTik\lib\x64
```

“库目录”中要添加 OpenCV 和 CUDA 的 lib 文件，示例路径为：

```
D:\OpenCV2.4.9\install_64\x64\vc10\lib
D:\CUDA6.5\CUDA_ToolTik\lib\x64
```

(4) 进入附加依赖项的配置，选择“链接器”→“输入”选项，右侧第一个就是“附加依赖项”选项，如图 5-30 所示。

在 Debug 和 Release 两个模式下添加附加依赖项有些区别。在 Debug 模式下添加：

```
cudart.lib
opencv_calib3d249d.lib
opencv_contrib249d.lib
opencv_core249d.lib
opencv_features2d249d.lib
opencv_flann249d.lib
opencv_gpu249d.lib
opencv_highgui249d.lib
opencv_imgproc249d.lib
opencv_legacy249d.lib
opencv_ml249d.lib
opencv_objdetect249d.lib
opencv_ts249d.lib
opencv_video249d.lib
```

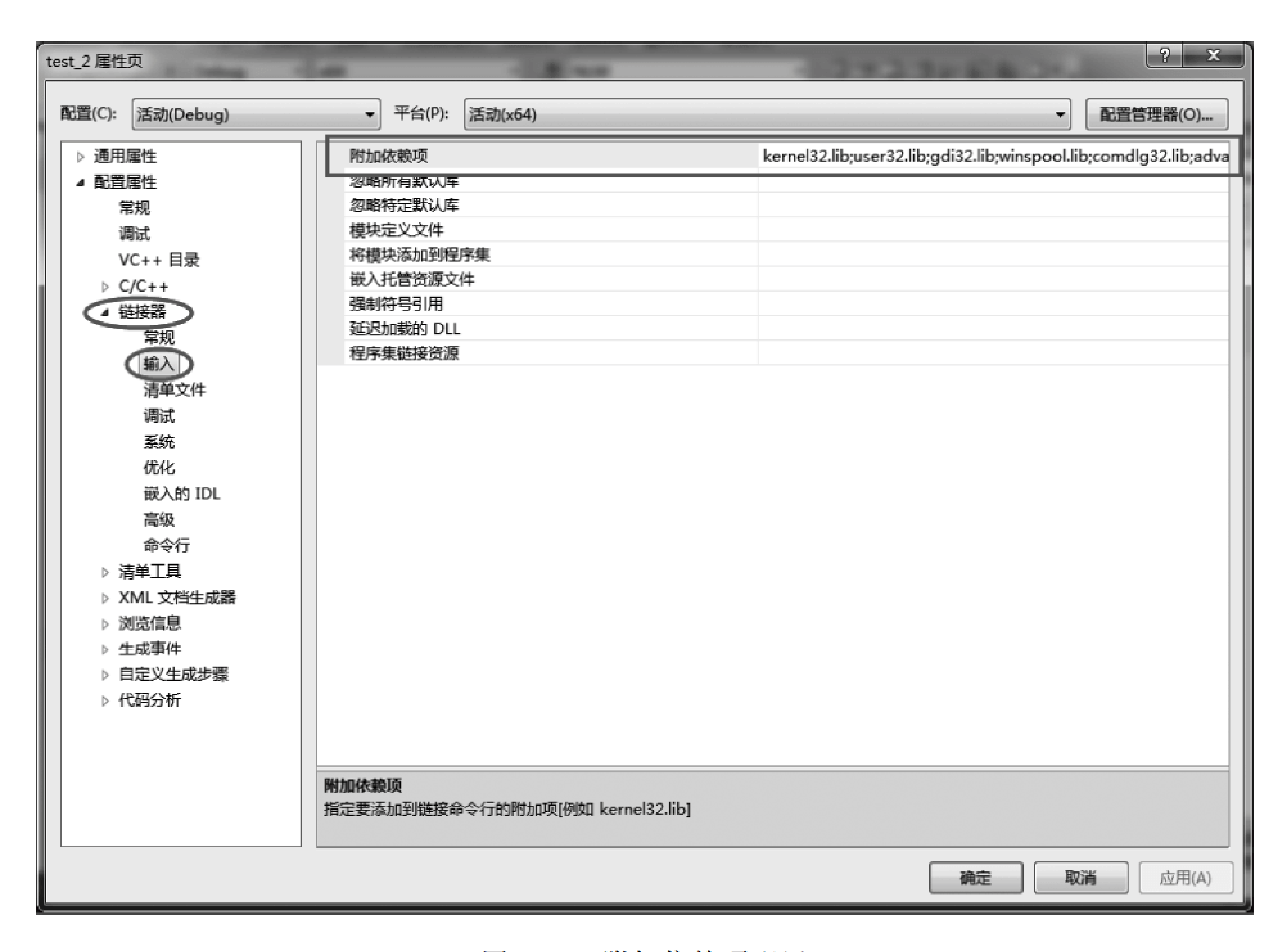


图 5-30 附加依赖项配置

在 Release 模式下需要添加：

```
cudart.lib
opencv_calib3d249.lib
opencv_contrib249.lib
opencv_core249.lib
opencv_features2d249.lib
opencv_flann249.lib
opencv_gpu249.lib
opencv_highgui249.lib
opencv_imgproc249.lib
opencv_legacy249.lib
opencv_ml249.lib
opencv_objdetect249.lib
opencv_ts249.lib
opencv_video249.lib
```

(5) 嵌入清单选项。回到属性界面，选择“清单工具”→“输入和输出”选项，并在右侧将“嵌入清单”选项改成“否”，如图 5-31 所示。

(6) 将属性页中的 Debug 改成 Release，如图 5-32 所示，重新进行配置，执行步骤(3)(4)(5)，但是请注意在“附加依赖项”里边 Debug 和 Release 所要填写的东西是不一样的。

(7) 回到主界面，右击项目文件，选择“生成自定义”命令，选中 CUDA 6.5 复选框即可，如图 5-33 所示。

至此，配置完成，可进入编程测试环节。

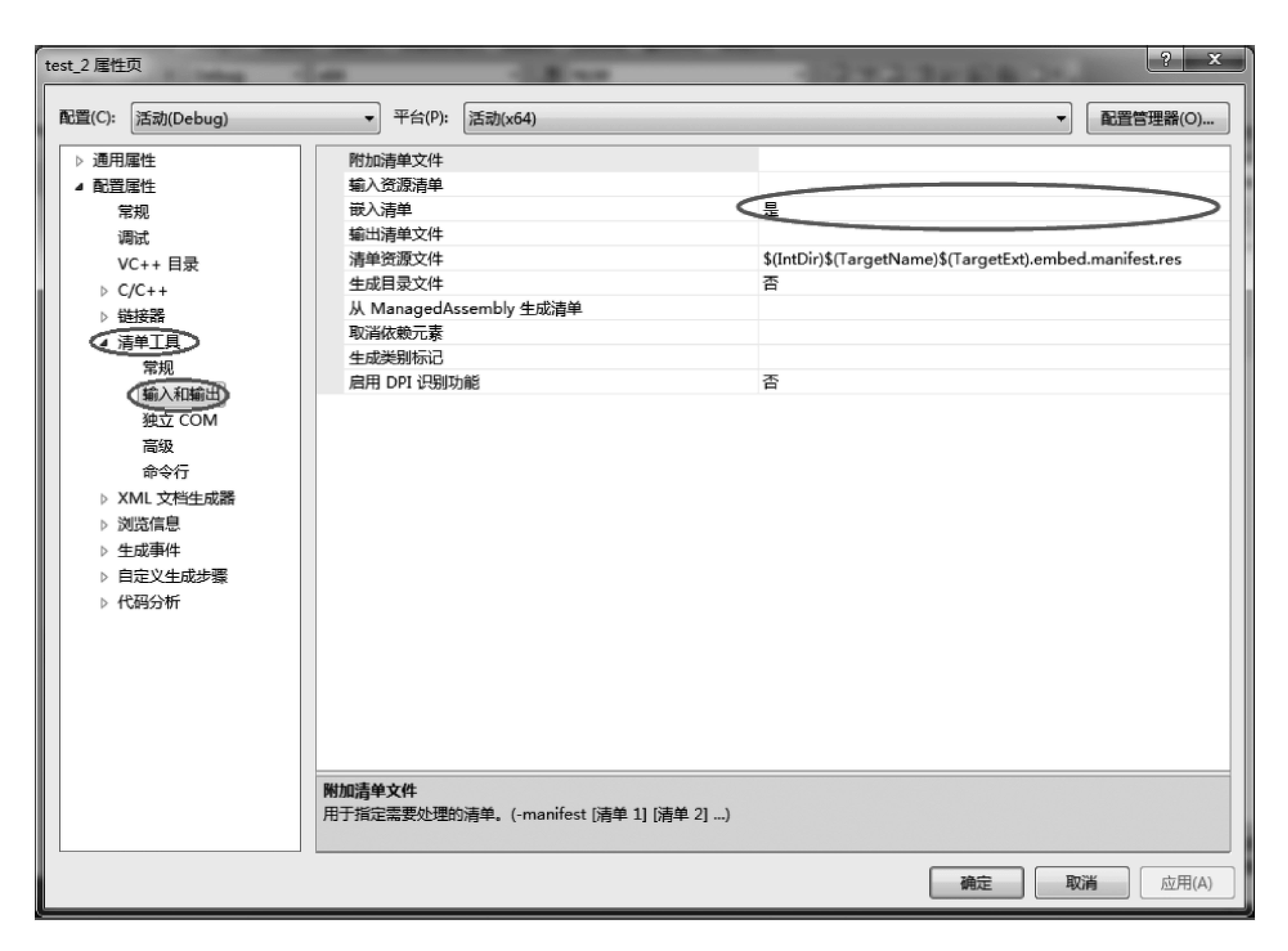


图 5-31 修改嵌入清单

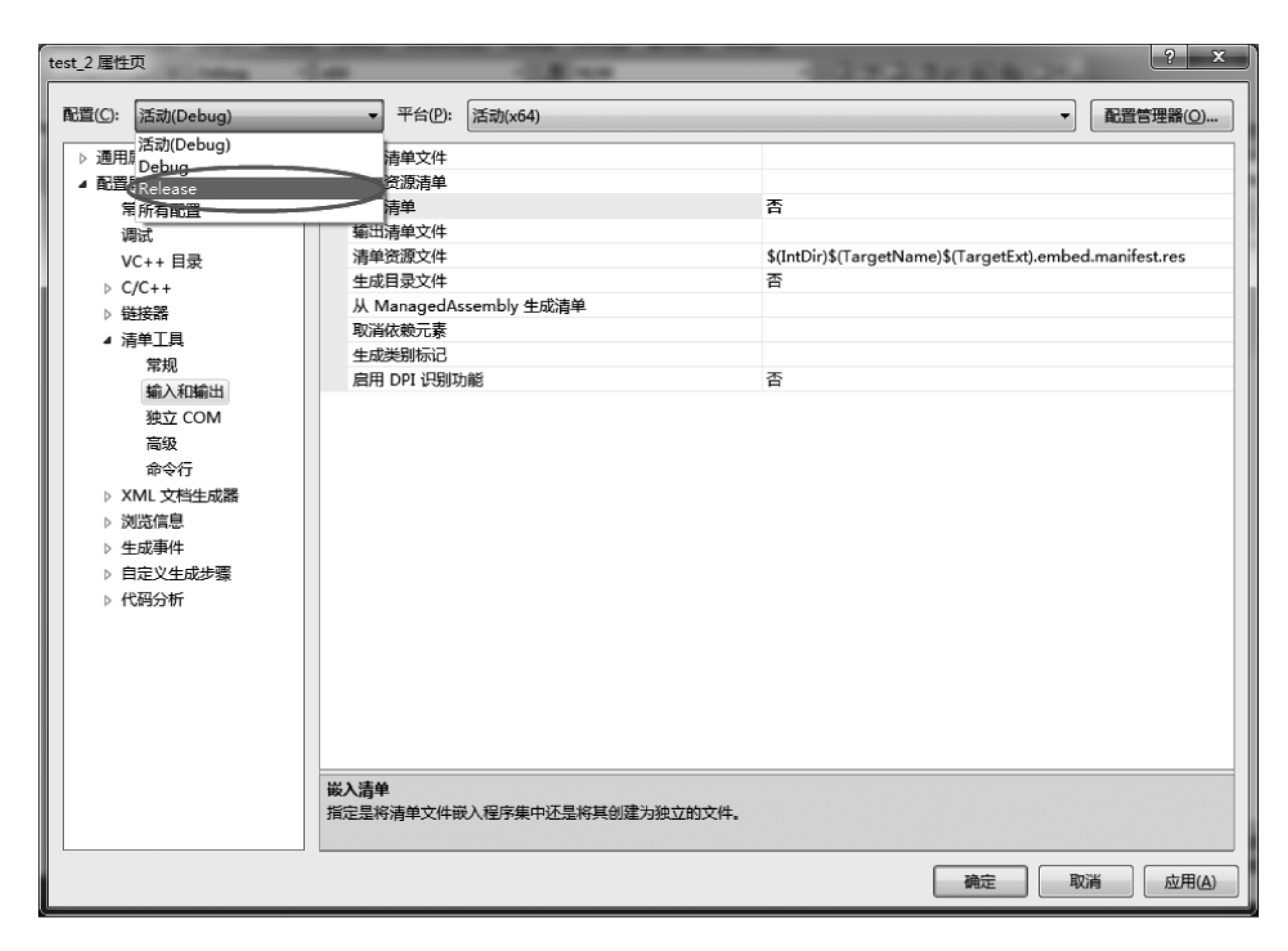


图 5-32 配置 Release 选项



图 5-33 生成自定义

2. 工程文件测试

本节将测试工程文件,使用的是与前面相同的反向算法。

反向算法的核心思想是用255减去灰度图的像素值,得到颜色翻转的图像:

$$g(x,y)=255-f(x,y) \tag{5-1}$$

在这里的反向算法中,为了方便测试,将使用单通道灰度图。其中.cpp 文件中存放控制 CPU 部分的程序,包括图像的读入、简单的预处理等等;而 GPU 部分主要负责计算任务,因此.cu 文件存放的是利用 GPU 实现遍历的算法,而.cuh 文件存放的是 CUDA 的头文件。

(1) 反向算法.cpp 文件。

```
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <cv.h>
#include <highgui.h>

#include <stdio.h>
#include <iostream>

#include "cuda.cuh"

using namespace cv;
using namespace std;
extern "C" void imghandle(uchar *ptr,int a,int b,int c);

int main( void )
{
    IplImage* imggray = cvLoadImage("lena256.jpg",0);
    cvNamedWindow("原始图像",CV_WINDOW_AUTOSIZE);
    cvShowImage("原始图像",imggray);
    double timeSpent = (double)getTickCount();
    int height = imggray->height;
```

```
    int width = imggray -> width;
    int step = imggray -> widthStep;
    int channels = imggray -> nChannels;
    int nwidthstep = imggray -> widthStep;
#define img_size height * width * nChannels
uchar *  imgdata = (uchar * )imggray -> imageData;
    printf("Processing a %dx%d image with %d channels\n",height,width,channels);
    imghandle(imgdata,height,width,nwidthstep);
    timeSpent = ((double)getTickCount() - timeSpent)/getTickFrequency();
cout << "Time spent in milliseconds: " << timeSpent * 1000 << endl;
    cvNamedWindow("反向图像",CV_WINDOW_AUTOSIZE);
    cvShowImage("反向图像",imggray);
    cvWaitKey(0);
}
```

(2) 反向算法的.cu 文件。

```
#include "cuda.cuh"
__global__ void imgreverse(unsigned char *dev_ptr,int height,int width,int nWidthStep)
{
    int ix = threadIdx.x + blockIdx.x * blockDim.x;
    int iy = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = ix + iy * nWidthStep;
    if(ix<width&&iy<height)
    {
        dev_ptr[offset] = 255 - dev_ptr[offset];
    }
}

extern "C"
    void imghandle(unsigned char *ptr,int height,int width,int nWidthStep)
{
    unsigned char *dev_ptr;
    cudaMalloc((void * * )&dev_ptr,256 * 256);
    cudaMemcpy(dev_ptr,ptr,256 * 256,cudaMemcpyHostToDevice);
    dim3 grids(16,16);
    dim3 threads(16,16);
    imgreverse<<<grids,threads>>>(dev_ptr,height,width,nWidthStep);
    cudaMemcpy(ptr,dev_ptr,256 * 256,cudaMemcpyDeviceToHost);
    cudaFree(dev_ptr);
}
```

(3)反向算法的.cuh 文件。

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>
```

(4) 运行结果。

如图 5-34 所示为运行结果,左侧是原图像,右侧是经过反向算法处理后的图像。

(5) 程序简析。

① 本次没有使用 OpenCV 2 系列中的 Mat 类,而是使用了 OpenCV 1 系列中的 cvLoadImage



图 5-34　测试结果

等函数,与 OpenCV 2 系列的函数相比其效率低、不够方便,同时还需要手动释放指针,不然就很容易出现指针越界等情况。虽然这种方法已经很少使用了,但为了更全面地学习,本次程序还是采用了 OpenCV 1 系列的函数。

② 在.cpp 文件中是无法直接调用.cu 文件中的非__global__型的函数的。如果想实现这种调用,就需要在.cu 文件中使用一个"extern "C"",以扩展一个 C 库的方式将.cpp 文件中的函数传送到.cu 文件中。

5.4　GPU 图像处理实例

本节将给出几个基础的示例,并在每个示例后给出程序的解释,以帮助入门级读者简略了解 GPU 图像处理的处理过程。如果需要更深入的学习,建议阅读 OpenCV 和 CUDA 的官方手册。

5.4.1　反向算法

反向算法在前面已经有所介绍,这里将处理三通道的彩色图像。本节以及后续几节的程序,都是直接在.cu 文件中实现的。

1. 程序实现

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_device_runtime_api.h>
#include <opencv2\gpu\gpu.hpp>
#include <opencv2\gpu\gpumat.hpp>
#include <opencv2\opencv.hpp>
#include <opencv.hpp>
#include <stdio.h>
#include <iostream>
#include <memory>
#include "opencv2/gpu/device/common.hpp"
#include "opencv2/gpu/device/reduce.hpp"
```

```
#include "opencv2/gpu/device/functional.hpp"
#include "opencv2/gpu/device/warp_shuffle.hpp"
using namespace std;
using namespace cv;
using namespace gpu;

template <int nthreads>
        __global__ void compute_kernel(int height, int width, const PtrStepb img,PtrStepb dst)
        {
            const int x = blockIdx.x * blockDim.x + threadIdx.x;    //x方向检索
            const int y = blockIdx.y * blockDim.y + threadIdx.y;    //y方向检索
            const uchar* src_y = (const uchar*)(img+y*img.step);    //原图像
            uchar* dst_y = (uchar*)(dst+y*dst.step);

            if (x<width && y<height)
            {
                dst_y[3*x] = 255-src_y[3*x];                        //三通道彩色图像处理
                dst_y[3*x+1] = 255-src_y[3*x+1];
                dst_y[3*x+2] = 255-src_y[3*x+2];
             }
        }

    int main()
{
    Mat a= imread("lena512.jpg");
    GpuMat d_a(a);                                              //GPUMat
    GpuMat d_dst(d_a.size(),CV_8UC3);
    int width = a.size().width;                                 //横向,x方向
    int height = a.size().height;                               //纵向,y方向

    const int nthreads =256;
    dim3 bdim(nthreads, 1);
    dim3 gdim(divUp(width, bdim.x), divUp(height, bdim.y));


    compute_kernel<nthreads><<<gdim, bdim>>>(height,width,d_a,d_dst);
    Mat dst(d_dst);
    imshow("原始图像",a);
    imshow("反向图像",dst);
    waitKey();
    return 0;
}
```

2. 运行结果

程序运行结果如图 5-35 所示。

3. 程序简析

1）GpuMat 类

GpuMat 类是 OpenCV 为 GPU 设计的 Mat 类变量。它仅可以在 GPU 内使用，相当于 CPU 中的 Mat 类，可以直接传递给 GPU。如果在 CPU 端使用 imshow 输出 GpuMat 类参



图 5-35　三通道图像反向算法运行结果

数或者是在 GPU 端输出 Mat 类参数则都会出错，使用方式如下：

```
GpuMat d_a(a);                          //将 Mat 类 a 赋给 GpuMat 类的 d_a
Mat dst(d_dst);                         //将 GPU 处理后的 GpuMat 类 d_dst 赋值给 Mat 类的 dst
GpuMat d_dst(d_a.size(),CV_8UC3);       //定义 GpuMat 类 d_dst,尺寸为 d_a,8 位三通道
```

GpuMat 类的详细数据结构如下：

- 它包含了下面的数据项。

--data：GPU 内存指针数据开始；

--step：距离之间的数据是两个连续的行；

--col，row：字段包含的图像大小；

--其他字段仅供内部使用。

- 内存分配。

```
void GpuMat::GpuMat(const cv::Size & size, int type);
void GpuMat::create(const cv::Size & size, int type);
```

内存分配应用示例：

```
GpuMat img( Size(1024,720), CV_8U );
GpuMat img2;
img2.create(Size(1,1000), CV_32FC3 );
```

其中：

CV_8U——单通道灰度图；

CV_8UC3——三通道 RGB 彩色图；

CV_32FC3——3D 立体点；

CV_16U——深度图。

- 为用户分配数据创建 GpuMat 数据头。

```
void GpuMat::GpuMat( const cv::Size& size, int type, void * data, size_t step);
trust::device_vector<float> vector(10000);
GpuMat header( Size(vector.size(),1),CV_32F, &vector[0], vector.size() * sizeof(float));
Cv::gpu::threshold(header, header,value, 1000, THRESH_BINARY );
```

- GPU-CPU 之间的数据传递。

```
void GpuMat::GpuMat(const cv::Mat& host_data);
void GpuMat::GpuMat(const cv::Mat& host_data);
void GpuMat::download(cv::Mat& host_data);
void GpuMat::copyTo(cv::gpu::GpuMat& other);
```

传递数据应用示例：

```
Mat host_image = cv::imload("lena512.jpg");          //读入图像
GpuMat device_image1;
device_image1.upload(host_image1);                   //分配内存并传到 GPU 中
GpuMat device_image2;
Device_image1.copyTo(device_image2);                 //分配内存并进行复制
Device_image2.download(host_image);                  //下载数据
```

2）线程数量

```
const int nthreads = 256;
dim3 bdim(nthreads, 1);
dim3 gdim(divUp(width, bdim.x), divUp(height, bdim.y));
```

为了保证线程的数量与像素点的数量相同，使用这种方式来划分线程还是很合理的。首先分配了一个 256×1 大小的线程块，之后每个线程块分线程时采用 divUp(width, bdim.x)函数进行分配。

divUp 是 OpenCV 内部的一个函数，其源码定义如下：

```
__host__ __device__ __forceinline__ int divUp(int total, int grain)
    {
        return (total + grain - 1) / grain;
    }
```

这样在调用线程块中的线程时，既可满足需要开启的线程数量要求，而且可以避免每一个线程块中的线程数量超出线程块内的上限问题。

例如，输入的图片是 512×512 个像素点，线程块的启动数量依旧为(256,1)个，而每个线程块中的线程就变成了(2,512)个，

$$(512+256-1)\div 256=2.996=2$$

$$(512+1-1)\div 1=512$$

通过这种方式可保证线程的数量大于等于像素点的数量，即确保每一个像素点可以分配到一个线程。当图像的像素点数量为 64×64 时，按照这个算法分配的线程块为(256,1)个，每个线程块中的线程为(1,64)。

3）if (x < width && y < height)

当启用的线程数量较多时，使用这个方式，就可以剔除超过像素点数量的线程，比如之前 64×64 个像素点的图片，在分配线程的时候启用了(256,1)(1,64)个线程，通过这种方式可以将超过图像大小的多余线程剔除掉，防止计算时出现错误。

4）三通道处理

```
dst_y[3 * x] = 255 - src_y[3 * x];
```

```
dst_y[3 * x + 1] = 255 - src_y[3 * x + 1];
dst_y[3 * x + 2] = 255 - src_y[3 * x + 2];
```

处理三通道图像时，图像是以 BGR 的形式进行存储的，也就是说，一个像素点会被分成三个连续的块。处理时需要使用 x 方向的索引，将其拆分开进行处理，因此第一行处理的是 B、第二行是 G、第三行是 R，最后将处理完成的像素点返回到 CPU，并进行输出。而在处理单通道的灰度图像时，读入仍然可以按照这种三通道方式进行读入，复制同样的图像信息在三通道中，经过 GPU 处理后进行返还，最后得到的图像也是灰度图处理后的图像，只是将被存为三通道灰度图，处理结果如图 5-36 所示。



图 5-36 单通道反向算法运行结果

5.4.2 图像加法、减法

1. 图像加法

图像的加法是图像处理中很常见同时也是很基础的算法，目标是将两张图叠加起来。通过 GPU 进行处理，使用之前介绍的反向算法的流程，只是将反向算法变换成了图像的叠加。图像加法的数学公式如下：

$$Z(x,y) = 0.5 \times f(x,y) + 0.5 \times g(x,y) \tag{5-2}$$

程序实现：

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_device_runtime_api.h>
#include <opencv2\gpu\gpu.hpp>
#include <opencv2\gpu\gpumat.hpp>
#include <opencv2\opencv.hpp>
#include <opencv.hpp>
#include <stdio.h>
#include <iostream>
#include <memory>
#include "opencv2/gpu/device/common.hpp"
#include "opencv2/gpu/device/reduce.hpp"
#include "opencv2/gpu/device/functional.hpp"
```

```
#include "opencv2/gpu/device/warp_shuffle.hpp"
using namespace std;
using namespace cv;
using namespace gpu;

template < int nthreads >
        __global__ void compute_kernel( int height, int width, const PtrStepb img, const
PtrStepb img2, PtrStepb dst)
        {
            const int x = blockIdx.x * blockDim.x + threadIdx.x;        //x 方向检索
            const int y = blockIdx.y * blockDim.y + threadIdx.y;        //y 方向检索


            const uchar * src_y = (const uchar * )(img + y * img.step);
            const uchar * src_y2 = (const uchar * )(img2 + y * img2.step);
            uchar * dst_y = (uchar * )(dst + y * dst.step);

            if (x < width && y < height)
            {
                dst_y[3 * x] = src_y[3 * x] * 0.5 + src_y2[3 * x] * 0.5;      //三通道图像
                dst_y[3 * x + 1] = src_y[3 * x + 1] * 0.5 + src_y2[3 * x + 1] * 0.5;
                dst_y[3 * x + 2] = src_y[3 * x + 2] * 0.5 + src_y2[3 * x + 2] * 0.5;
             }
        }

    int main()
{
    Mat b = imread("wood.jpg");
    Mat a = imread("rain.jpg");

    GpuMat d_a(a);
    GpuMat d_b(b);

    GpuMat d_dst(d_a.size(),CV_8UC3);

    int width = a.size().width;
    int height = a.size().height;

    const int nthreads = 256;
    dim3 bdim(nthreads, 1);
    dim3 gdim(divUp(width, bdim.x), divUp(height, bdim.y));

    compute_kernel < nthreads ><<< gdim, bdim >>>(height,width,d_a,d_b,d_dst);
    Mat dst(d_dst);
    imshow("原始图像 a",a);
    imshow("原始图像 b",b);
    imshow("叠加后图像",dst);
    waitKey();
    return 0;
}
```

运行结果如图 5-37 所示。

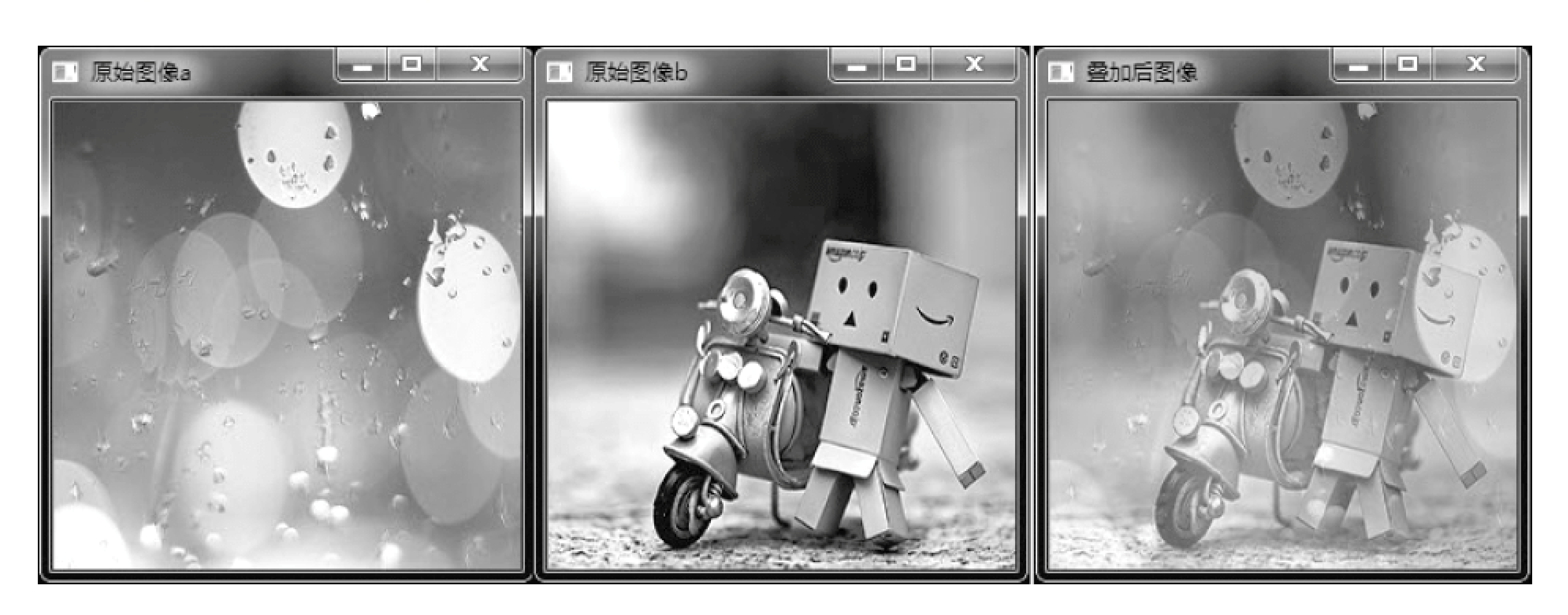


图 5-37 图像加法

2. 图像减法

图像减法与图像加法相反，是图像中点对点的相减，通常用于去掉图像中的相同部分，得到图像中有差异的部分。但是图像减法需要注意像素点的值小于 0 的情况，因此可以使用绝对值的方式或者使用其他方式灵活地得到图像之间的差异。

图像减法的数学公式如下：

$$Z(x,y)=|f(x,y)-g(x,y)| \tag{5-3}$$

因为多数图像中，相似的部分很多，做差之后为纯黑色，即像素点的值为 0。因此，为了效果更明显一些，可以在相减结果基础上增加一个反向算法使其更容易观察：

$$Z(x,y)=255-|f(x,y)-g(x,y)| \tag{5-4}$$

程序实现：

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_device_runtime_api.h>
#include <opencv2\gpu\gpu.hpp>
#include <opencv2\gpu\gpumat.hpp>
#include <opencv2\opencv.hpp>
#include <opencv.hpp>
#include <stdio.h>
#include <iostream>
#include <memory>
#include "opencv2/gpu/device/common.hpp"
#include "opencv2/gpu/device/reduce.hpp"
#include "opencv2/gpu/device/functional.hpp"
#include "opencv2/gpu/device/warp_shuffle.hpp"
using namespace std;
using namespace cv;
using namespace gpu;

template <int nthreads>
        __global__ void compute_kernel(int height, int width, const PtrStepb img, const
PtrStepb img2,PtrStepb dst)
```

```
        {
            const int x = blockIdx.x * blockDim.x + threadIdx.x;
            const int y = blockIdx.y * blockDim.y + threadIdx.y;

            const uchar* src_y = (const uchar*)(img + y * img.step);
            const uchar* src_y2 = (const uchar*)(img2 + y * img2.step);
            uchar* dst_y = (uchar*)(dst + y * dst.step);

            if (x < width && y < height)
            {
              dst_y[3 * x] = 255 - abs(src_y[3 * x] - src_y2[3 * x]);
              dst_y[3 * x + 1] = 255 - abs(src_y[3 * x + 1] - src_y2[3 * x + 1]);
              dst_y[3 * x + 2] = 255 - abs(src_y[3 * x + 2] - src_y2[3 * x + 2]);

            }
        }

    int main()
{
    Mat b = imread("vessel.jpg");
    Mat a = imread("vessel2.jpg");

    GpuMat d_a(a);
    GpuMat d_b(b);
    GpuMat d_dst(d_a.size(),CV_8UC3);

    int width = a.size().width;
    int height = a.size().height;

    const int nthreads = 256;
    dim3 bdim(nthreads, 1);
    dim3 gdim(divUp(width, bdim.x), divUp(height, bdim.y));

    compute_kernel<nthreads><<<gdim, bdim>>>(height,width,d_a,d_b,d_dst);
    Mat dst(d_dst);
    imshow("原始图像 a",a);
    imshow("原始图像 b",b);
    imshow("相减后的图像",dst);
    waitKey();
    return 0;
}
```

运行结果如图 5-38 所示,左上角和右上角为差异图像,左下角为相减后的图像,右下角为相减后反向的结果。

5.4.3 图像腐蚀、膨胀

图像的腐蚀和膨胀是图像数学形态学的基础之一,图像腐蚀(Erosion)可以使一个孤立的低亮噪音扩大化,同时也可以将物体的一些低亮度的关键细节丢失。图像膨胀(Dilation)可以使一个孤立的高亮噪声扩大化,也可以使物体的一些高亮度噪声的关键细节丢失[2]。

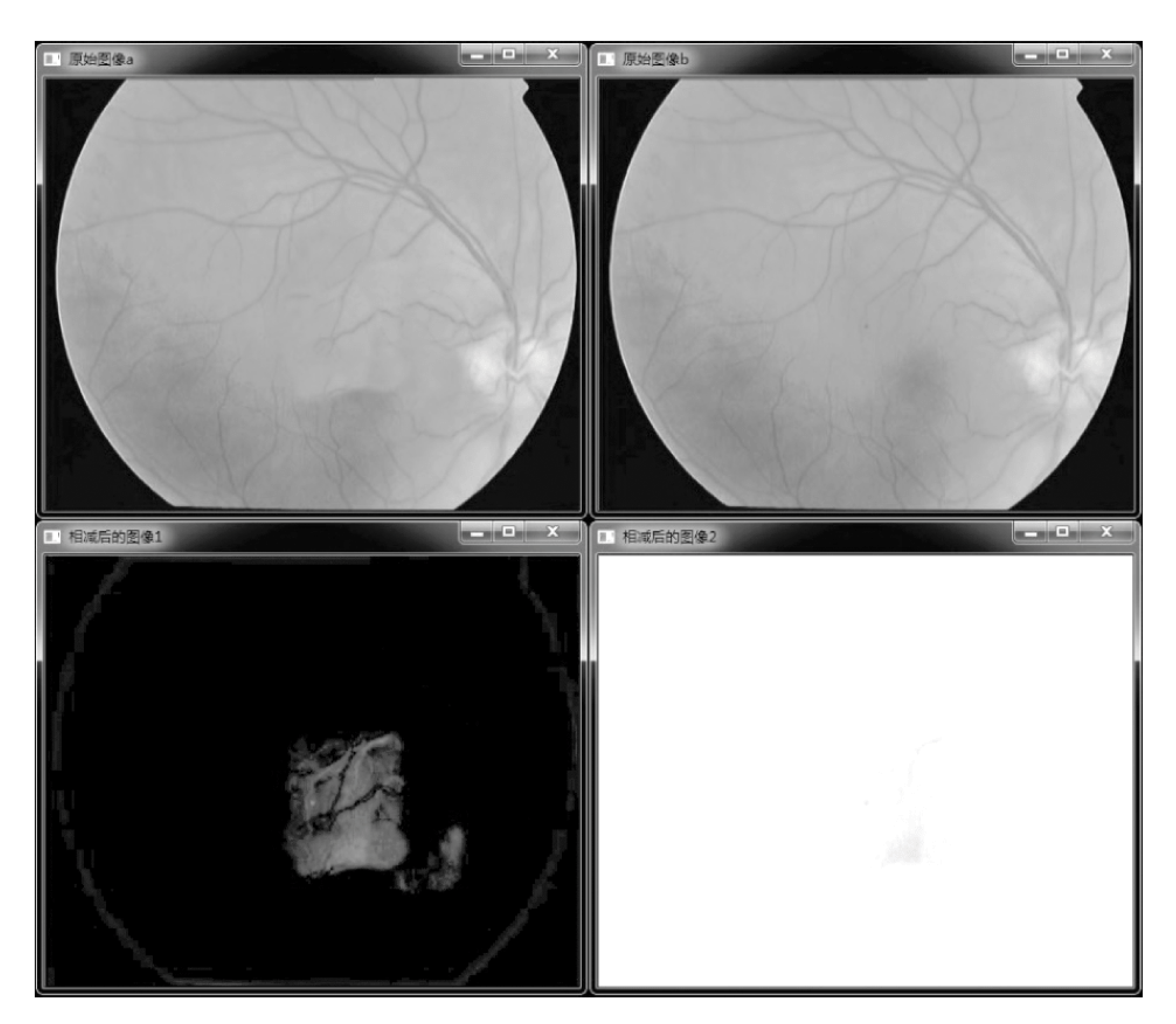
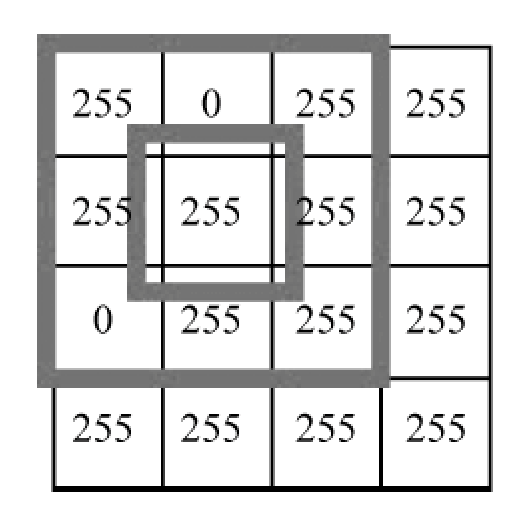


图 5-38　图像相减

1. 二值图像的腐蚀和膨胀

1）二值图像腐蚀

在图像中，取一个 3×3 的像素块，中间点的四周如果有像素点的值为 0，则将中心点的像素值变为 0，如图 5-39 所示。



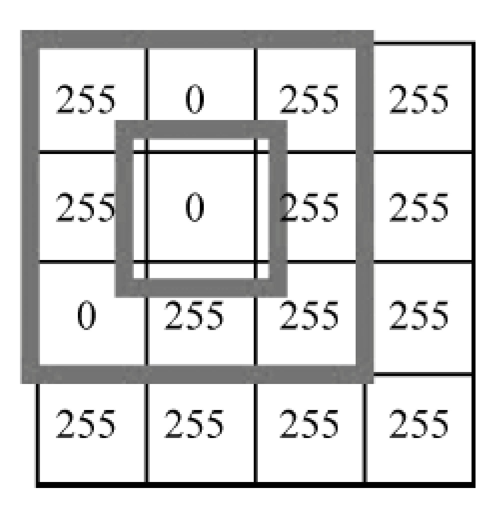


图 5-39　二值腐蚀的原理

程序实现：

首先将原始的彩色图像读入，转换成单通道的灰度图像，在将灰度图像传入 GPU 中，先变换为二值图像，最后对二值图像实现图像腐蚀的算法。

```
#include "cuda_runtime.h"
```

```
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_device_runtime_api.h>
#include <opencv2\gpu\gpu.hpp>
#include <opencv2\gpu\gpumat.hpp>
#include <opencv2\opencv.hpp>
#include <opencv.hpp>
#include <stdio.h>
#include <iostream>
#include <memory>
#include <cv.h>
#include <highgui.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include "opencv2/gpu/device/common.hpp"
#include "opencv2/gpu/device/reduce.hpp"
#include <opencv2/highgui/highgui.hpp>
#include "opencv2/gpu/device/functional.hpp"
#include "opencv2/gpu/device/warp_shuffle.hpp"
using namespace std;
using namespace cv;
using namespace gpu;

template <int nthreads>
        __global__ void compute_kernel(int height, int width, const PtrStepb img,PtrStepb dst)
        {
            const int x = blockIdx.x * blockDim.x + threadIdx.x;         //x索引
            const int y = blockIdx.y * blockDim.y + threadIdx.y;         //y索引
//图像二值化
            const uchar * src_yy = (const uchar * )(img + y * img.step);
            uchar * dst_yy = (uchar * )(dst + y * dst.step);

            if(x < width && y < height)
            {
                if( src_yy[x]>= 128 )
                    dst_yy[x] = 255;
                else
                    dst_yy[x] = 0;
            }

//二值化图像实现图像腐蚀
            const uchar * src_y = (const uchar * )(img);
            uchar * dst_y = (uchar * )(dst);

            int x_1 = max(0,x-1);                      //用来找到图像四周的像素点
            int x1 = min(width-1,x+1);
            int y_1 = max(0,y-1);
            int y1 = min(height-1,y+1);

            if (x < width && y < height)
            {
```

```
                if(
                    src_y[y_1 * img.step + x_1] == 0 ||
                    src_y[y_1 * img.step + x] == 0 ||
                    src_y[y_1 * img.step + x1] == 0 ||
                    src_y[y * img.step + x_1] == 0 ||
                    src_y[y * img.step + x] == 0 ||
                    src_y[y1 * img.step + x_1] == 0 ||
                    src_y[y * img.step + x] == 0 ||
                    src_y[y * img.step + x1] == 0 )
                {
                    dst[y * img.step + x]  =  0;
                }
            }

        }


    int main()
{
    Mat src =  imread("lena256.jpg",1);
    Mat a =  imread("lena256.jpg",0);

    int b = a.channels();
    printf("读入图图像通道 = %d\n ",b);


    GpuMat d_a(a);
    GpuMat d_dst(a);
    int width  =  a.size().width;
    int height  =  a.size().height;

    const int nthreads  = 256;
    dim3 bdim(nthreads, 1);                           //确定线程数量
    dim3 gdim(divUp(width, bdim.x), divUp(height, bdim.y));
    compute_kernel<nthreads><<<gdim, bdim>>>(height,width,d_a,d_dst);


    Mat dst(d_dst);
    imshow("原始图像",src);
    imshow("单通道图像",a);
    imshow("图像腐蚀后的 i 效果图",dst);
    waitKey();
    return 0;
}
```

运行结果如图 5-40 所示。

2）二值图像膨胀

膨胀与腐蚀相反，在图像中，取一个 3×3 的像素块，中间点的四周如果有像素点的值为 255，则将中心点的像素值变为 255，如图 5-41 所示。



图 5-40　二值腐蚀

255	0	255	255
255	0	255	255
0	255	255	255
255	255	255	255

255	0	255	255
255	255	255	255
0	255	255	255
255	255	255	255

图 5-41　二值膨胀的原理

程序实现：

首先将原始的彩色图像读入，转换成单通道的灰度图像，再将灰度图像传入 GPU 中，先变换为二值图像，最后通过二值图像实现图像膨胀算法。

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_device_runtime_api.h>
#include <opencv2\gpu\gpu.hpp>
#include <opencv2\gpu\gpumat.hpp>
#include <opencv2\opencv.hpp>
#include <opencv.hpp>
#include <stdio.h>
#include <iostream>
#include <memory>
#include <cv.h>
```

```
#include <highgui.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include "opencv2/gpu/device/common.hpp"
#include "opencv2/gpu/device/reduce.hpp"
#include <opencv2/highgui/highgui.hpp>
#include "opencv2/gpu/device/functional.hpp"
#include "opencv2/gpu/device/warp_shuffle.hpp"
using namespace std;
using namespace cv;
using namespace gpu;

template <int nthreads>
        __global__ void compute_kernel(int height, int width, const PtrStepb img,PtrStepb dst)
        {
//图像二值化
            const int ix = blockIdx.x * blockDim.x + threadIdx.x;
            const int iy = blockIdx.y * blockDim.y + threadIdx.y;
            const uchar * src_yy = (const uchar *)(img + iy * img.step);
            uchar * dst_yy = (uchar *)(dst + iy * dst.step);

            if(ix < width && iy < height)
            {
                if( src_yy[ix]>= 128 )
                    dst_yy[ix] = 255;
                else
                    dst_yy[ix] = 0;
            }


//利用二值图实现图像膨胀
            const uchar * src_y = (const uchar *)(img);
            uchar * dst_y = (uchar *)(dst);


            int ix_1 = max(0,ix - 1);
            int ix1 = min(width - 1,ix + 1);
            int iy_1 = max(0,iy - 1);
            int iy1 = min(height - 1,iy + 1);

            if (ix < width && iy < height)
            {
                if(
                    src_y[iy_1 * img.step + ix_1] == 0 ||
                    src_y[iy_1 * img.step + ix] == 0 ||
                    src_y[iy_1 * img.step + ix1] == 0 ||
                    src_y[iy * img.step + ix_1] == 0 ||
                    src_y[iy * img.step + ix] == 0 ||
                    src_y[iy1 * img.step + ix_1] == 0 ||
                    src_y[iy * img.step + ix] == 0 ||
                    src_y[iy * img.step + ix1] == 0 )
                {
                    dst[iy * img.step + ix] = 255;
                }
            }

        }
```

```
    int main()
{
    Mat src = imread("lena256.jpg",1);
    Mat a = imread("lena256.jpg",0);

    int b = a.channels();
    printf("读入图图像通道 = %d\n",b);

    GpuMat d_a(a);
    GpuMat d_dst(a);
    int width = a.size().width;
    int height = a.size().height;

    const int nthreads = 256;
    dim3 bdim(nthreads, 1);
    dim3 gdim(divUp(width, bdim.x), divUp(height, bdim.y));
    compute_kernel<nthreads><<<gdim, bdim>>>(height,width,d_a,d_dst);


    Mat dst(d_dst);
    imshow("原图像",src);
    imshow("单通道图像",a);
    imshow("图像膨胀后的效果图",dst);
    waitKey();
    return 0;
}
```

实现结果如图 5-42 所示。



图 5-42　二值图像膨胀

2. 灰度图像腐蚀和膨胀

1）灰度图像腐蚀

与二值图像腐蚀的原理一样，在灰度图的情况下，图像的腐蚀就是将一个确定范围内的最小值赋给中心位置的像素点，如图 5-43 所示。

209	125	191	9	168	52	65
33	234	162	44	203	128	235
121	150	55	66	32	87	231
32	76	244	121	141	122	136
14	158	191	246	196	98	204
235	8	1	71	126	235	66

209	125	191	9	168	52	65
33	234	162	44	203	128	235
121	150	55	66	32	87	231
32	76	244	121	32	122	136
14	158	191	246	196	98	204
235	8	1	71	126	235	66

图 5-43　灰度图像腐蚀原理

程序实现：

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_device_runtime_api.h>
#include <opencv2\gpu\gpu.hpp>
#include <opencv2\gpu\gpumat.hpp>
#include <opencv2\opencv.hpp>
#include <opencv.hpp>
#include <stdio.h>
#include <iostream>
#include <memory>
#include <cv.h>
#include <highgui.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include "opencv2/gpu/device/common.hpp"
#include "opencv2/gpu/device/reduce.hpp"
#include <opencv2/highgui/highgui.hpp>
#include "opencv2/gpu/device/functional.hpp"
#include "opencv2/gpu/device/warp_shuffle.hpp"
using namespace std;
using namespace cv;
using namespace gpu;

template <int nthreads>
        __global__ void compute_kernel(int height, int width, const PtrStepb img,PtrStepb dst)
        {
            const int ix = blockIdx.x * blockDim.x + threadIdx.x; //x
            const int iy = blockIdx.y * blockDim.y + threadIdx.y; //y

            const uchar * src_y = (const uchar * )(img);
            uchar * dst_y = (uchar * )(dst);
```

```
            int ix_1 = max(0,ix - 1);
            int ix1 = min(width - 1,ix + 1);
            int iy_1 = max(0,iy - 1);
            int iy1 = min(height - 1,iy + 1);

            if (ix < width && iy < height)
            {
                if( src_y[iy * img.step + ix] > src_y[iy_1 * img.step + ix_1] ) dst_y[iy * img.
step + ix] = src_y[iy_1 * img.step + ix_1];
                if( src_y[iy * img.step + ix] > src_y[iy_1 * img.step + ix] ) dst_y[iy * img.
step + ix] = src_y[iy_1 * img.step + ix];
                if( src_y[iy * img.step + ix] > src_y[iy_1 * img.step + ix1] ) dst_y[iy * img.
step + ix] = src_y[iy_1 * img.step + ix1];
                if( src_y[iy * img.step + ix] > src_y[iy * img.step + ix_1] ) dst_y[iy * img.
step + ix] = src_y[iy * img.step + ix_1];
                if( src_y[iy * img.step + ix] > src_y[iy * img.step + ix] ) dst_y[iy * img.step +
ix] = src_y[iy * img.step + ix];
                if( src_y[iy * img.step + ix] > src_y[iy1 * img.step + ix_1] ) dst_y[iy * img.
step + ix] = src_y[iy1 * img.step + ix_1];
                if( src_y[iy * img.step + ix] > src_y[iy * img.step + ix] ) dst_y[iy * img.step +
ix] = src_y[iy * img.step + ix];
                if( src_y[iy * img.step + ix] > src_y[iy * img.step + ix1] ) dst_y[iy * img.
step + ix] = src_y[iy * img.step + ix1];

            }
        }


    int main()
{
      Mat src = imread("lena256.jpg",1);
    Mat a = imread("lena256.jpg",0);

    int b = a.channels();
    printf("读入图像的通道数 = %d\n",b);


    GpuMat d_a(a);
    GpuMat d_dst(a);
    int width = a.size().width;                         //横向
    int height = a.size().height;                       //纵向

    const int nthreads = 256;
    dim3 bdim(nthreads, 1);                             //分配线程
    dim3 gdim(divUp(width, bdim.x), divUp(height, bdim.y));
    compute_kernel<nthreads><<<gdim, bdim>>>(height,width,d_a,d_dst);

    Mat dst(d_dst);
    imshow("原图像",src);
    imshow("单通道图像",a);
    imshow("图像腐蚀后的效果图",dst);
```

```
    waitKey();
    return 0;
}
```

运行结果如图 5-44 所示。



图 5-44　单通道图像腐蚀

2）灰度图像膨胀

单通道的图像膨胀与单通道的图像腐蚀相反，就是将一个确定范围内的最大值赋给中心位置的像素点，如图 5-45 所示。

209	125	191	9	168	52	65
33	234	162	44	203	128	235
121	150	55	66	32	87	231
32	76	244	121	141	122	136
14	158	191	246	196	98	204
235	8	1	71	126	235	66

209	125	191	9	168	52	65
33	234	162	44	203	128	235
121	150	55	66	32	87	231
32	76	244	121	246	122	136
14	158	191	246	196	98	204
235	8	1	71	126	235	66

图 5-45　单通道图像膨胀

程序实现：

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_device_runtime_api.h>
#include <opencv2\gpu\gpu.hpp>
#include <opencv2\gpu\gpumat.hpp>
#include <opencv2\opencv.hpp>
#include <opencv.hpp>
#include <stdio.h>
#include <iostream>
#include <memory>
#include <cv.h>
#include <highgui.h>
#include <opencv2/core/core.hpp>
```

```
#include <opencv2/imgproc/imgproc.hpp>
#include "opencv2/gpu/device/common.hpp"
#include "opencv2/gpu/device/reduce.hpp"
#include <opencv2/highgui/highgui.hpp>
#include "opencv2/gpu/device/functional.hpp"
#include "opencv2/gpu/device/warp_shuffle.hpp"
using namespace std;
using namespace cv;
using namespace gpu;

template <int nthreads>
        __global__ void compute_kernel(int height, int width, const PtrStepb img,PtrStepb dst)
        {
            const int ix = blockIdx.x * blockDim.x + threadIdx.x; //x
            const int iy = blockIdx.y * blockDim.y + threadIdx.y; //y

            const uchar * src_y = (const uchar * )(img);
            uchar * dst_y = (uchar * )(dst);


            int ix_1 = max(0,ix-1);
            int ix1 = min(width-1,ix+1);
            int iy_1 = max(0,iy-1);
            int iy1 = min(height-1,iy+1);

            if (ix < width && iy < height)
            {
                if( src_y[iy * img.step + ix] <= src_y[iy_1 * img.step + ix_1] ) dst_y[iy *
img.step + ix] = src_y[iy_1 * img.step + ix_1];
                if( src_y[iy * img.step + ix] <= src_y[iy_1 * img.step + ix] ) dst_y[iy * img.
step + ix] = src_y[iy_1 * img.step + ix];
                if( src_y[iy * img.step + ix] <= src_y[iy_1 * img.step + ix1] ) dst_y[iy *
img.step + ix] = src_y[iy_1 * img.step + ix1];
                if( src_y[iy * img.step + ix] <= src_y[iy * img.step + ix_1] ) dst_y[iy * img.
step + ix] = src_y[iy * img.step + ix_1];
                if( src_y[iy * img.step + ix] <= src_y[iy * img.step + ix] ) dst_y[iy * img.
step + ix] = src_y[iy * img.step + ix];
                if( src_y[iy * img.step + ix] <= src_y[iy1 * img.step + ix_1] ) dst_y[iy *
img.step + ix] = src_y[iy1 * img.step + ix_1];
                if( src_y[iy * img.step + ix] <= src_y[iy * img.step + ix] ) dst_y[iy * img.
step + ix] = src_y[iy * img.step + ix];
                if( src_y[iy * img.step + ix] <= src_y[iy * img.step + ix1] ) dst_y[iy * img.
step + ix] = src_y[iy * img.step + ix1];

            }
        }


    int main()
{
    Mat src = imread("lena256.jpg",1);
```

```
    Mat a = imread("lena256.jpg",0);

    int b = a.channels();
    printf("读入图像通道数 = %d\n",b);


    GpuMat d_a(a);
    GpuMat d_dst(a);
    int width  = a.size().width;                     //横向
    int height  = a.size().height;                   //纵向

    const int nthreads  = 256;
    dim3 bdim(nthreads, 1);                          //分配线程
    dim3 gdim(divUp(width, bdim.x), divUp(height, bdim.y));
    compute_kernel<nthreads><<<gdim, bdim>>>(height,width,d_a,d_dst);

    Mat dst(d_dst);
    imshow("原图像",src);
    imshow("单通道图像",a);
    imshow("图像腐蚀后的效果图",dst);
    waitKey();
    return 0;
}
```

运行结果如图 5-46 所示。



图 5-46　单通道图像膨胀

5.4.4　非局部均值算法

非局部均值(Non-Local Means，NLM)算法是一种去噪效果很好的滤波算法。

1. 原理

1) 邻域平均去噪

邻域平均去噪是非局部均值滤波的理论基础。假设图像受到加性高斯白噪声的干扰，那么可以得到被干扰后的图像：

$$g(x)=f(x)+\varphi(x) \tag{5-5}$$

其中 $f(x)$表示原本的纯净图像，$g(x)$表示受到干扰之后的输出图像，$\varphi(x)$表示均值为零

的高白斯噪声。

邻域平均去噪是利用邻域像素的均值估计中心像素点，属于较早的一种去噪策略。该方法是基于这样一种前提假设，噪声在图像局部区域内服从相同的分布，并且像素点的灰度值在非常小的范围内是缓慢变化的，即一定程度上是相似的。因此，可以用邻域像素估计中心像素的值。对于一个给定的像素点 i，设 $N(i)$ 为所选取的用于平均计算的邻域，则像素 $f(i)$ 的估计值 $\hat{f}(i)$ 为：

$$\hat{f}(i)=\frac{1}{N}\sum_{j\in N(i)}(f(j)+\varphi(j))=\frac{1}{N}\sum_{j\in N(i)}f(j)+\frac{1}{N}\sum_{j\in N(i)}\varphi(j) \tag{5-6}$$

式中 N 表示 $N(i)$ 内像素点的个数，因 $E[\varphi(j)]=0$，若 $f(i)=f(j)$ 则有 $\hat{f}(i)=\hat{f}(j)$。用 VA 表示像素点 i 去噪后的方差，σ^2 为噪声信号 $\varphi(j)$ 的方差，则有：

$$\mathrm{VA}=\mathrm{Var}\left\{\frac{1}{N}\sum_{j\in N(i)}\varphi(j)\right\}=\frac{1}{N^2}\sum_{j\in N(i)}\mathrm{Var}[\varphi(j)]=\frac{1}{N^2}N\sigma^2=\frac{1}{N}\sigma^2 \tag{5-7}$$

由上式可知，N 值越大，滤波后的像素 i 处的噪声方差就会越小，仅为原来的 $\frac{1}{N}$。

在实际去噪过程中，由于噪声的污染，在图像中确定 $f(i)$ 的真实值是比较困难的，并且噪声较大时去噪能力有限。虽然完全相同的像素较少，但是在一定邻域内，相似的像素却有很多。为了能够充分利用这一特性，学者们提出了加权平均的思想。

加权平均是利用图像中的自相似信息，根据像素之间的相似程度设置权值的大小。基于加权平均的邻域平均去噪算法取得了非常好的去噪效果，此算法的关键在于如何度量像素之间的相似性或者构造权值函数，这就是非局部均值算法的前身。

2）非局部均值算法

局部去噪和变换域去噪算法在去除噪声的同时，能够恢复图像的主要几何结构信息，但在精细结构、细节信息和纹理的保留方面明显不足。图像中的任何一个像素都不是孤立的，而是与其周围的像素点结合在一起共同构成图形的几何结构。以某一个像素点为中心的窗口邻域，可以很好地描述像素点的结构特征。针对任何一个像素点的图像块的所有集合可以看作是图像的一种过完备表示。它采用结构相似性定义像素间的差异，并对像素周围整个区域的灰度分布做整体对比，根据图像中灰度分布的相似性决定权值的大小，如图 5-47 所示，假设 p、q_1、q_2、q_3 具有完全相同的灰度值，那么 q_1、q_2、q_3 三者都会根据与 p 点不同的欧氏距离而得到相应的权值，但 q_1 和 q_2 的邻域灰度分布与 p 更接近，因此贡献更大的权值，q_3 则对 p 贡献较小的权值。



图 5-47 相似图像块示意图

假设一幅含噪图像 $z=\{z(i)\mid i\in I\}$，其定义在有界域 $I\in N^2$。在这幅图像中，对于某个像素点 i，非局部均值滤波算法利用所有像素的加权平均来得到该点的估计值 $NL(z)(i)$，即：

$$NL(z)(i)=\sum_{j\in I}w(i,j)z(j) \tag{5-8}$$

其中，权值$\{w(i,j)\}_j$依赖于像素i与像素j之间的相似性，且满足如下条件：$0\leqslant w(i,j)\leqslant 1$且$\sum_j w(i,j)=1$。

图像域I上的邻域系统$N=\{N_i\}_{i\in I}$是图像域I的子集，使得对于所有的像素点$i\in I$都必须要满足以下两个条件：

(1) $i\in N_i$；

(2) $j\in N_i \Rightarrow i\in N_j$；

其中，N_i是像素i的窗口邻域，N_j是像素j的窗口邻域。

为了能够更好地适应图像不同区域的特征，可以将相似性窗口取不同的形状和大小。为了方便起见，此处使用固定大小的方形窗口。相似性窗口N_i内的灰度值向量可以表示如下：

$$z(N_i)=(z(j),j\in N_i) \tag{5-9}$$

灰度值向量$z(N_i)$和$z(N_j)$之间的相似性可以用来决定像素点i和像素点j之间的相似性，即在加权平均时，那些与$z(N_i)$具有相似灰度值向量的像素点将被分配较大的权值，反之则被分配到较小的权值。为了能够定量地计算$z(N_i)$和$z(N_j)$之间的相似性，可以采用高斯加权的欧氏距离$\| z(N_i)-z(N_j)\|_{2,a}^2$。含噪声的图像与滤波后的图像在对应位置上的相似性窗口内灰度值向量之间的欧氏距离满足如下的关系：

$$E\| z(N_i)-z(N_j)\|_{2,a}^2=\| y(N_i)-y(N_j)\|_{2,a}^2+2\sigma^2 \tag{5-10}$$

其中，z与y分别表示带有噪声图像与滤波后的图像，σ^2是噪声的方差。

基于以上的式子可以得到像素点i和像素点j之间的权值$w(i,j)$：

$$w(i,j)=\frac{1}{Z(i)}\exp\left(\frac{-\| z(N_i)-z(N_j)\|_{2,a}^2}{h^2}\right) \tag{5-11}$$

其中$Z(i)=\sum_j \exp\left(\frac{-\| z(N_i)-z(N_j)\|_{2,a}^2}{h^2}\right)$是归一化常数；$\| z(N_i)-z(N_j)\|_{2,a}^2$是指$i$块和$j$块的加权欧式距离的平方，这里用$d(i,j)$来表示，$a$ ($a>0$)是指高斯核的标准差，由选定像素邻域的窗口大小决定。参数h控制指数函数的衰减速度，同时影响着权值的衰减速度，$h=c\times\sigma$。c是用于调整的系数，Buades将其范围规定在1～10之间。

最终可以将非局部均值滤波的算法整理为如下三个式子：

$$d(i,j)=\| z(N_i)-z(N_j)\|_{2,a}^2 \tag{5-12}$$

$$w(i,j)=\exp\left(-\frac{d(i,j)}{h^2}\right) \tag{5-13}$$

$$z(i)=\frac{\sum_{j\in I} w(i,j)z(j)}{\sum_{j\in I} w(i,j)} \tag{5-14}$$

图5-48显示了非局部均值滤波算法的执行过程。在算法执行的过程中，需要设置两个窗口的大小：一个是像素邻域窗口尺寸$K\times K$；另一个是像素邻域窗口搜索范围的窗口尺寸$L\times L$，即在$L\times L$大小的窗口内选择像素的邻域大小为$K\times K$执行非局部均值滤波算法，$K\times K$的窗口在$L\times L$的区域内滑动，根据区域的相似性确定区域中心像素灰度所贡献的权值，而在这里的图像处理实现中，窗口尺寸为$K=7$，滑动范围为$L=17$。

在相对稳定的条件下，即图像有足够大的尺寸，对于图像内部的各种细节都能找到足够多的相似区域。

2. 程序实现

这套程序结合了 CUDA 6.5 给出的示例和一些 OpenGL 函数，加上 OpenCV 共同实现。程序由五个部分组成，可以处理三通道的彩色图像，如图 5-49 所示。

- bmploader.cpp 的主体功能是确认导入的图片是否为.bmp 格式的图像。
- imageDenoising.cu 包含了一些简单的预处理部分和将数据传到.cuh 文件中做处理。
- imageDenoising.h 中包含了函数中的起始变量和一些头文件。
- imageDenoising_nlm2_kernel.cuh 中主要放置了 GPU 实现 NLM 算法的部分。
- imageDenoisingGL.cpp 为主函数。

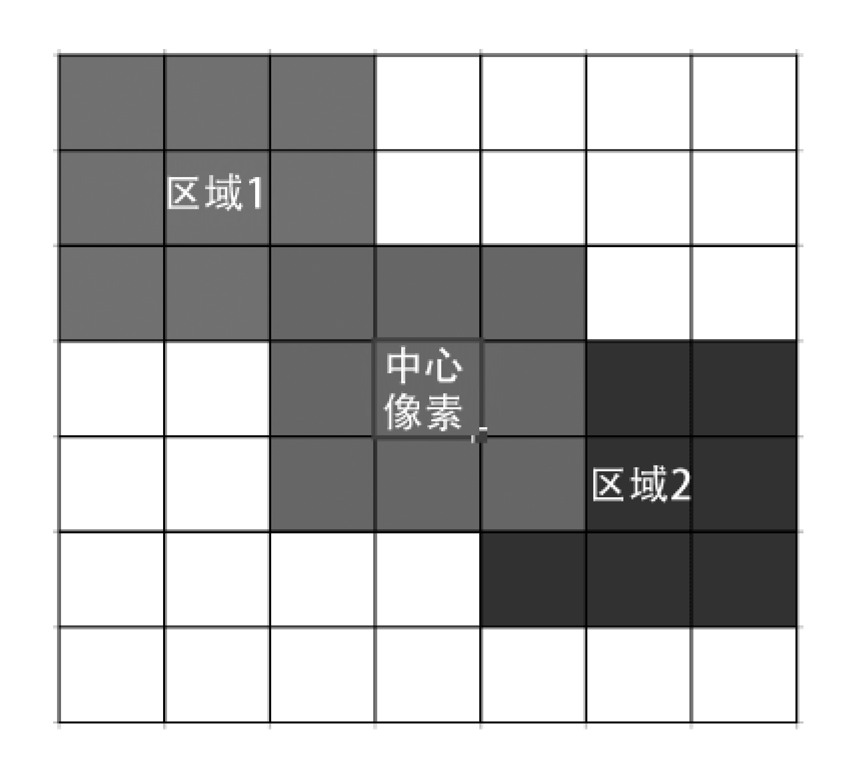


图 5-48 非局部均值滤波算法执行示意图



图 5-49 NLM 工程文件

程序如下：

(1) bmploader.cpp

```
#include <stdio.h>
#include <stdlib.h>

#if defined(WIN32) || defined(_WIN32) || defined(WIN64) || defined(_WIN64)
# pragma warning( disable : 4996 )                    //disable deprecated warning
#endif

#pragma pack(1)

typedef struct
{
    short type;
    int size;
    short reserved1;
    short reserved2;
    int offset;
} BMPHeader;

typedef struct
{
    int size;
```

```
    int width;
    int height;
    short planes;
    short bitsPerPixel;
    unsigned compression;
    unsigned imageSize;
    int xPelsPerMeter;
    int yPelsPerMeter;
    int clrUsed;
    int clrImportant;
} BMPInfoHeader;


//Isolated definition
typedef struct
{
    unsigned char x, y, z, w;
} uchar4;


/* ------------------ 确定读入的是 BMP 格式的图像 ----------------- */
extern "C" void LoadBMPFile(uchar4 **dst, int *width, int *height, const char *name)
{
    BMPHeader hdr;
    BMPInfoHeader infoHdr;
    int x, y;

    FILE *fd;
    printf("Loading %s...\n", name);
    if (sizeof(uchar4) != 4)
    {
        printf("***Bad uchar4 size***\n");
        exit(EXIT_SUCCESS);
    }

    if (!(fd = fopen(name,"rb")))
    {
        printf("***BMP load error: file access denied***\n");
        exit(EXIT_SUCCESS);
    }

    fread(&hdr, sizeof(hdr), 1, fd);

    if (hdr.type != 0x4D42)
    {
        printf("***BMP load error: bad file format***\n");
        exit(EXIT_SUCCESS);
    }
```

```
        fread(&infoHdr, sizeof(infoHdr), 1, fd);

        if (infoHdr.bitsPerPixel != 24)
        {
            printf("***BMP load error: invalid color depth***\n");
            exit(EXIT_SUCCESS);
        }

        if (infoHdr.compression)
        {
            printf("***BMP load error: compressed image***\n");
            exit(EXIT_SUCCESS);
        }

        *width = infoHdr.width;                       //读入宽度
        *height = infoHdr.height;                     //读入高度
        *dst = (uchar4 *)malloc(*width * *height * 4);

        printf("BMP width: %u\n", infoHdr.width);
        printf("BMP height: %u\n", infoHdr.height);

        fseek(fd, hdr.offset - sizeof(hdr) - sizeof(infoHdr), SEEK_CUR);

        for (y = 0; y < infoHdr.height; y++)
        {
            for (x = 0; x < infoHdr.width; x++)
            {
                (*dst)[(y * infoHdr.width + x)].z = fgetc(fd);
                (*dst)[(y * infoHdr.width + x)].y = fgetc(fd);
                (*dst)[(y * infoHdr.width + x)].x = fgetc(fd);
            }

            for (x = 0; x < (4 - (3 * infoHdr.width) % 4) % 4; x++)
                fgetc(fd);
        }
        if (ferror(fd))
        {
            printf("***Unknown BMP load error.***\n");
            free(*dst);
            exit(EXIT_SUCCESS);
        }
        else
            printf("BMP file loaded successfully!\n");
        fclose(fd);
    }
```

(2) imageDenoising.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include "imageDenoising.h"
#include "imageDenoising_nlm2_kernel.cuh"


////////////////////////////////////////////////////////////////////////////////
//调用的简单函数
////////////////////////////////////////////////////////////////////////////////
float Max(float x, float y)
{
    return (x > y) ? x : y;
}

float Min(float x, float y)
{
    return (x < y) ? x : y;
}

int iDivUp(int a, int b)
{
    return ((a % b) != 0) ? (a / b + 1) : (a / b);
}

__device__ float lerpf(float a, float b, float c)
{
    return a + (b - a) * c;
}

__device__ float vecLen(float4 a, float4 b)
{
    return (
                (b.x - a.x) * (b.x - a.x) +
                (b.y - a.y) * (b.y - a.y) +
                (b.z - a.z) * (b.z - a.z)
            );
}

__device__ TColor make_color(float r, float g, float b, float a)
{
    return
        ((int)(a * 255.0f) << 24) |
        ((int)(b * 255.0f) << 16) |
        ((int)(g * 255.0f) << 8) |
        ((int)(r * 255.0f) << 0);
}



////////////////////////////////////////////////////////////////////////////////
//Global data handlers and parameters
////////////////////////////////////////////////////////////////////////////////
//Texture reference and channel descriptor for image texture
```

```
texture<uchar4, 2, cudaReadModeNormalizedFloat> texImage;
cudaChannelFormatDesc uchar4tex = cudaCreateChannelDesc<uchar4>();

//CUDA array descriptor
cudaArray *a_Src;

////////////////////////////////////////////////////////////////////////////////
//Filtering kernels
////////////////////////////////////////////////////////////////////////////////

extern "C"
cudaError_t CUDA_Bind2TextureArray()
{
    return cudaBindTextureToArray(texImage, a_Src);
}

extern "C"
cudaError_t CUDA_UnbindTexture()
{
    return cudaUnbindTexture(texImage);
}

extern "C"
cudaError_t CUDA_MallocArray(uchar4 **h_Src, int imageW, int imageH)
{
    cudaError_t error;

    error = cudaMallocArray(&a_Src, &uchar4tex, imageW, imageH);
    error = cudaMemcpyToArray(a_Src, 0, 0,
                              *h_Src, imageW * imageH * sizeof(uchar4),
                              cudaMemcpyHostToDevice
                              );

    return error;
}


extern "C"
cudaError_t CUDA_FreeArray()
{
    return cudaFreeArray(a_Src);
}
```

(3) imageDenoising.h

```
/*--------------定义噪声的头文件-------------*/

#ifndef IMAGE_DENOISING_H
#define IMAGE_DENOISING_H
typedef unsigned int TColor;
```

```
/* ---------------------------------- 初始化数据 ------------------------ */
#define KNN_WINDOW_RADIUS 3
#define NLM_WINDOW_RADIUS 3
#define NLM_BLOCK_RADIUS 3
#define KNN_WINDOW_AREA ( (2 * KNN_WINDOW_RADIUS + 1) * (2 * KNN_WINDOW_RADIUS + 1) )
#define NLM_WINDOW_AREA ( (2 * NLM_WINDOW_RADIUS + 1) * (2 * NLM_WINDOW_RADIUS + 1) )
#define INV_KNN_WINDOW_AREA ( 1.0f / (float)KNN_WINDOW_AREA )
#define INV_NLM_WINDOW_AREA ( 1.0f / (float)NLM_WINDOW_AREA )
#define KNN_WEIGHT_THRESHOLD 0.02f
#define KNN_LERP_THRESHOLD 0.79f
#define NLM_WEIGHT_THRESHOLD 0.10f
#define NLM_LERP_THRESHOLD 0.10f
#define BLOCKDIM_X 8
#define BLOCKDIM_Y 8
#ifndef MAX
#define MAX(a,b) ((a < b) ? b : a)
#endif
#ifndef MIN
#define MIN(a,b) ((a < b) ? a : b)
#endif

/* ------------------------ 图像读入 ---------------------- */
extern "C" void LoadBMPFile(uchar4 ** dst, int *width, int *height, const char *name);
extern "C" cudaError_t CUDA_Bind2TextureArray();
extern "C" cudaError_t CUDA_UnbindTexture();
extern "C" cudaError_t CUDA_MallocArray(uchar4 ** h_Src, int imageW, int imageH);
extern "C" cudaError_t CUDA_FreeArray();
extern "C" void cuda_NLM2(TColor *d_dst, int imageW, int imageH, float Noise, float LerpC);
extern "C" void cuda_NLM2diag(TColor * d_dst, int imageW, int imageH, float Noise, float
LerpC);

#endif
```

(4) imageDenoising_nlm2_kernel.cuh

```
__global__ void NLM2(
    TColor *dst,
    int imageW,
    int imageH,
    float Noise,
    float lerpC
)
{
    __shared__ float fWeights[BLOCKDIM_X * BLOCKDIM_Y];
    const int ix = blockDim.x * blockIdx.x + threadIdx.x;
    const int iy = blockDim.y * blockIdx.y + threadIdx.y;

    const float x = (float)ix + 0.5f;
    const float y = (float)iy + 0.5f;
    const float cx = blockDim.x * blockIdx.x + NLM_WINDOW_RADIUS + 0.5f;
    const float cy = blockDim.x * blockIdx.y + NLM_WINDOW_RADIUS + 0.5f;
```

```
if (ix < imageW && iy < imageH)
{
    //for 循环得到权重
    float weight = 0;
    for (float n = - NLM_BLOCK_RADIUS; n <= NLM_BLOCK_RADIUS; n++)
        for (float m = - NLM_BLOCK_RADIUS; m <= NLM_BLOCK_RADIUS; m++)
            weight += vecLen(
                          tex2D(texImage, cx + m, cy + n),
                          tex2D(texImage, x + m, y + n)
                     );

    float dist =
        (threadIdx.x - NLM_WINDOW_RADIUS) * (threadIdx.x - NLM_WINDOW_RADIUS) +
        (threadIdx.y - NLM_WINDOW_RADIUS) * (threadIdx.y - NLM_WINDOW_RADIUS);

    weight = __expf( - (weight * Noise + dist * INV_NLM_WINDOW_AREA));


    fWeights[threadIdx.y * BLOCKDIM_X + threadIdx.x] = weight;

    __syncthreads();

    //NLM 的标准计数器
    float fCount = 0;
    float sumWeights = 0;

    //结果累加
    float3 clr = {0, 0, 0};

    int idx = 0;
    for (float i = - NLM_WINDOW_RADIUS; i <= NLM_WINDOW_RADIUS + 1; i++)
        for (float j = - NLM_WINDOW_RADIUS; j <= NLM_WINDOW_RADIUS + 1; j++)
        {
            float weightIJ = fWeights[idx++];
            float4 clrIJ = tex2D(texImage, x + j, y + i);
            clr.x += clrIJ.x * weightIJ;
            clr.y += clrIJ.y * weightIJ;
            clr.z += clrIJ.z * weightIJ;

            //权重叠加 w(i,j)
            sumWeights += weightIJ;
        fCount += (weightIJ > NLM_WEIGHT_THRESHOLD) ? INV_NLM_WINDOW_AREA : 0;
        }


    sumWeights = 1.0f / sumWeights;
    clr.x *= sumWeights;                            //处理后的图像
    clr.y *= sumWeights;
    clr.z *= sumWeights;
```

```
        float lerpQ = (fCount > NLM_LERP_THRESHOLD) ? lerpC : 1.0f - lerpC;
        float4 clr00 = tex2D(texImage, x, y);
        clr.x = lerpf(clr.x, clr00.x, lerpQ);
        clr.y = lerpf(clr.y, clr00.y, lerpQ);
        clr.z = lerpf(clr.z, clr00.z, lerpQ);
        dst[imageW * iy + ix] = make_color(clr.x, clr.y, clr.z, 0);
    }
}

extern "C"
void cuda_NLM2(
    TColor *d_dst,
    int imageW,
    int imageH,
    float Noise,
    float LerpC
)
{
    dim3 threads(BLOCKDIM_X, BLOCKDIM_Y);
    dim3 grid(iDivUp(imageW, BLOCKDIM_X), iDivUp(imageH, BLOCKDIM_Y));

    NLM2<<<grid, threads>>>(d_dst, imageW, imageH, Noise, LerpC);
}
```

(5) imageDenoisingGL.cpp

```
#include <opencv2/core/core.hpp>                        //OpenCV 库
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>

#include <GL/glew.h>                                    //OpenGL 库
#if defined(__APPLE__) || defined(MACOSX)
#include <GLUT/glut.h>
#else
#include <GL/freeglut.h>
#endif

#include <cuda_runtime.h>                               //CUDA 头文件
#include <cuda_gl_interop.h>

#include <stdio.h>                                      //C++输入输出头文件
#include <stdlib.h>
#include <string.h>
#include "imageDenoising.h"

#include <helper_functions.h>
#include <helper_cuda.h>
using namespace std;

const char *sSDKsample = "CUDA ImageDenoising";
```

```
/* --------------------- 滤波模式的选择,这里仅有 NLM 一种 ------------------- */
const char * filterMode[] =
{
    "Passthrough",
    "KNN method",
    "NLM method",
    "Quick NLM(NLM2) method",
    NULL
};

/* -------------- 确认图像是否生效 --------------- */
const char * sOriginal[] =
{
    "image_passthru.ppm",
    "image_knn.ppm",
    "image_nlm.ppm",
    "image_nlm2.ppm",
    NULL
};

const char * sReference[] =
{
    "ref_passthru.ppm",
    "ref_knn.ppm",
    "ref_nlm.ppm",
    "ref_nlm2.ppm",
    NULL
};


GLuint gl_PBO, gl_Tex;
struct cudaGraphicsResource * cuda_pbo_resource;
uchar4 * h_Src;
int imageW, imageH;                                    //图像宽度、高度
GLuint shader;

//初始化
int  g_Kernel = 0;
bool  g_FPS = false;
bool  g_Diag = false;
StopWatchInterface * timer = NULL;

const float noiseStep = 0.025f;
const float lerpStep = 0.025f;
static float knnNoise = 0.32f;
static float nlmNoise = 1.45f;
static float lerpC = 0.2f;


const int frameN = 24;
int frameCounter = 0;
```

```
#define BUFFER_DATA(i) ((char *)0 + i)

const int frameCheckNumber = 4;
int fpsCount = 0;
int fpsLimit = 1;
unsigned int frameCount = 0;
unsigned int g_TotalErrors = 0;

int *pArgc = NULL;
char **pArgv = NULL;

#define MAX_EPSILON_ERROR 5
#define REFRESH_DELAY 10                    //时间单位 10ms

void computeFPS()
{
    frameCount++;
    fpsCount++;

    if (fpsCount == fpsLimit)
    {
        char fps[256];
        float ifps = 1.f / (sdkGetAverageTimerValue(&timer) / 1000.f);
        sprintf(fps, "<%s>: %3.1f fps", filterMode[g_Kernel], ifps);

        glutSetWindowTitle(fps);
        fpsCount = 0;
        sdkResetTimer(&timer);
    }
}

/*------------------ 选择模块(这里仅有 NLM 一种) -------------------*/
void runImageFilters(TColor *d_dst)
{
    if(g_Kernel = 1)
    {
        cuda_NLM2(d_dst, imageW, imageH, 1.0f / (nlmNoise * nlmNoise), lerpC);
    }
}


/*---------------- 显示图片的模块测试图片是否出错也在这里 ------------------*/
void displayFunc(void)
{
    sdkStartTimer(&timer);                   //SDK 初始化
    TColor *d_dst = NULL;
    size_t num_bytes;
    if (frameCounter++ == 0)
    {
        sdkResetTimer(&timer);
    }
```

```
            //测试 GPU 内存
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_pbo_resource, 0));
    getLastCudaError("cudaGraphicsMapResources failed");
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&d_dst, &num_bytes,
cuda_pbo_resource));
    getLastCudaError("cudaGraphicsResourceGetMappedPointer failed");
    checkCudaErrors(CUDA_Bind2TextureArray());

    runImageFilters(d_dst);                        //runImageFilters 函数中做选择效果

    checkCudaErrors(CUDA_UnbindTexture());
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_pbo_resource, 0));

    //正常的显示初始化和显示效果
    {
        glClear(GL_COLOR_BUFFER_BIT);

        glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, imageW, imageH, GL_RGBA, GL_UNSIGNED_BYTE,
BUFFER_DATA(0));
        glBegin(GL_TRIANGLES);
        glTexCoord2f(0, 0);
        glVertex2f(-1, -1);
        glTexCoord2f(2, 0);
        glVertex2f(+3, -1);
        glTexCoord2f(0, 2);
        glVertex2f(-1, +3);
        glEnd();
        glFinish();
    }

    if (frameCounter == frameN)
    {
        frameCounter = 0;

        if (g_FPS)                                 //显示出 FPS
        {
            printf("FPS: %3.1f\n", frameN / (sdkGetTimerValue(&timer) * 0.001));
            g_FPS = false;
        }
    }

    glutSwapBuffers();

    sdkStopTimer(&timer);
    computeFPS();
}

void timerEvent(int value)
{
    glutPostRedisplay();
    glutTimerFunc(REFRESH_DELAY, timerEvent, 0);
```

```
}

void shutDown(unsigned char k, int /*x*/, int /*y*/)
{
    switch (k)
    {
        case 'q':
        case 'Q':
            printf("Shutting down...\n");
            sdkStopTimer(&timer);
            sdkDeleteTimer(&timer);
            checkCudaErrors(CUDA_FreeArray());
            free(h_Src);
            exit(EXIT_SUCCESS);                    //成功即退出
            break;
    }
}

/*--------------图像GL初始化----------*/
int initGL(int *argc, char **argv)
{
    glutInit(argc, argv);                          //将数据送入GL允许其使用
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);  //GL显示
    glutInitWindowSize(imageW, imageH);
    glutInitWindowPosition(512 - imageW / 2, 384 - imageH / 2);
                                                   //窗口位置
    glutCreateWindow(argv[0]);
    glewInit();                                    //GL初始化完成
    return 0;
}


static const char *shader_code =
    "!!ARBfp1.0\n"
    "TEX result.color, fragment.texcoord, texture[0], 2D; \n"
    "END";


GLuint compileASMShader(GLenum program_type, const char *code) {
    GLuint program_id;
    glGenProgramsARB(1, &program_id);
    glBindProgramARB(program_type, program_id);
    glProgramStringARB(program_type, GL_PROGRAM_FORMAT_ASCII_ARB, (GLsizei) strlen(code),
(GLubyte *) code);

    GLint error_pos;
    glGetIntegerv(GL_PROGRAM_ERROR_POSITION_ARB, &error_pos);

    if (error_pos != -1)
    {
        const GLubyte *error_string;
```

```
            error_string = glGetString(GL_PROGRAM_ERROR_STRING_ARB);
            fprintf(stderr, "Program error at position: %d\n%s\n", (int)error_pos, error_string);
            return 0;
        }

        return program_id;
    }


    void initOpenGLBuffers()
    {
        glEnable(GL_TEXTURE_2D);
        glGenTextures(1, &gl_Tex);
        glBindTexture(GL_TEXTURE_2D, gl_Tex);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, imageW, imageH, 0, GL_RGBA, GL_UNSIGNED_BYTE, h_Src);
        printf("Texture created.\n");
        printf("Creating PBO...\n");
        glGenBuffers(1, &gl_PBO);
        glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, gl_PBO);
        glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB, imageW * imageH * 4, h_Src, GL_STREAM_COPY);
        checkCudaErrors ( cudaGraphicsGLRegisterBuffer ( &cuda _ pbo _ resource, gl _ PBO,
    cudaGraphicsMapFlagsWriteDiscard));
        GLenum gl_error = glGetError();

        if (gl_error != GL_NO_ERROR)
        {
    #if defined(WIN32) || defined(_WIN32) || defined(WIN64) || defined(_WIN64)
                          //判断系统的位数
            char tmpStr[512];
            sprintf_s(tmpStr, 255, "\n%s(%i) : GL Error : %s\n\n", __FILE__, __LINE__,
    gluErrorString(gl_error));
            OutputDebugString(tmpStr);
    #endif
            fprintf(stderr, "GL Error in file '%s' in line %d :\n", __FILE__, __LINE__);
            fprintf(stderr, "%s\n", gluErrorString(gl_error));
            exit(EXIT_FAILURE);
        }
        printf("PBO created.\n");
        shader = compileASMShader(GL_FRAGMENT_PROGRAM_ARB, shader_code);
    }


    /* -------------------- 清理函数 ---------------- */
    void cleanup()
    {
        sdkDeleteTimer(&timer);
        glDeleteProgramsARB(1, &shader);
```

```
}

void runAutoTest(int argc, char * * argv, const char * filename, int kernel_param)
{
    printf("[ % s] - (automated testing w/ readback)\n", sSDKsample);

    int devID = findCudaDevice(argc, (const char * * )argv);


    printf("Allocating host and CUDA memory and loading image file...\n");

    const char * image_path = sdkFindFilePath("barbara.png", argv[0]);

    if (image_path == NULL)
    {
        printf("imageDenoisingGL was unable to find and load image file <portrait_noise.bmp>.\
nExiting...\n");
        exit(EXIT_FAILURE);
    }

    LoadBMPFile(&h_Src, &imageW, &imageH, image_path);
    printf("Data init done.\n");

    checkCudaErrors(CUDA_MallocArray(&h_Src, imageW, imageH));

    TColor * d_dst = NULL;
    unsigned char * h_dst = NULL;
    checkCudaErrors(cudaMalloc((void * * )&d_dst, imageW * imageH * sizeof(TColor)));
    h_dst = (unsigned char * )malloc(imageH * imageW * 4);

    {
        g_Kernel = kernel_param;
        printf("[AutoTest]: % s <% s>\n", sSDKsample, filterMode[g_Kernel]);
        checkCudaErrors(CUDA_Bind2TextureArray());
        runImageFilters(d_dst);
        checkCudaErrors(CUDA_UnbindTexture());
        checkCudaErrors(cudaDeviceSynchronize());

        checkCudaErrors(cudaMemcpy(h_dst, d_dst, imageW * imageH * sizeof(TColor),
cudaMemcpyDeviceToHost));
        sdkSavePPM4ub(filename, h_dst, imageW, imageH);
    }

    checkCudaErrors(CUDA_FreeArray());
    free(h_Src);

    checkCudaErrors(cudaFree(d_dst));
    free(h_dst);

    printf("\n[ % s] -> Kernel % d, Saved: % s\n", sSDKsample, kernel_param, filename);
    cudaDeviceReset();
```

```
    exit(g_TotalErrors == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}



/* ---------------------------- 主函数 ----------------------------------- */
int main(int argc, char * * argv)
{
    char * dump_file = NULL;                        //图像指针
    pArgc = &argc;                                  //读取数据进行初始化
    pArgv = argv;

    //if 语句用来初始化检测 GPU 应用情况
    if (checkCmdLineFlag(argc, (const char * * )argv, "file"))
            //使用 CMD 去指令去测试原始的 GPU 能否使用,不能使用则做初始化
    {
        getCmdLineArgumentString(argc, (const char * * )argv,"file", (char * * ) &dump_file);
        int kernel = 1;
        if (checkCmdLineFlag(argc, (const char * * )argv, "kernel"))
        {
            kernel = getCmdLineArgumentInt(argc, (const char * * )argv, "kernel");
        }
        runAutoTest(argc, argv, dump_file, kernel);
    }

    else
    {
/* -------------------------- 读入图像 --------------------------------- */
      const char * image_path = sdkFindFilePath("lena512_noise.bmp", argv[0]);
/* ------------------------------------------------------------------- */

        if (image_path == NULL)
        {
            exit(EXIT_FAILURE);
        }

        LoadBMPFile(&h_Src, &imageW, &imageH, image_path);     //检查图像格式

    }


/* ---------------------- 计时函数 -------------------- */
double timeSpent1 = (double)cv::getTickCount();
/* -------------------------------------------------- */

    initGL(&argc, argv);                            //将图像信息送入 initGL 做图像的初始化
    cudaGLSetGLDevice(gpuGetMaxGflopsDeviceId());   //初始化 CUDAGL,helper 文件
    checkCudaErrors(CUDA_MallocArray(&h_Src, imageW, imageH));
                            //在 imageDenoising.cu 中写入图像的数据、宽度和高度
    initOpenGLBuffers();
```

```
    g_Kernel = 1;                              //直接进入 NLM 算法
    glutDisplayFunc(displayFunc);              //显示功能界面
    glutKeyboardFunc(shutDown);
    sdkCreateTimer(&timer);                    //时间函数
    sdkStartTimer(&timer);

/* ------------------------ 计时代码 ------------------------ */
timeSpent1 = ((double)cv::getTickCount() - timeSpent1)/cv::getTickFrequency();
    cout << "NLM 算法实现耗时"<< endl;
    cout << "Time spent in milliseconds: " << timeSpent1 * 1000 << endl;
/* ------------------------------------------------------------- */
    glutMainLoop();
    cudaDeviceReset();                         //重置
    exit(EXIT_SUCCESS);                        //退出
}
```

3. 运行结果

运行结果如图 5-50 所示，左侧图像是原始图像，中间的图像是加高斯白噪声之后的图像，而右侧的图像是经过处理后的图像。



图 5-50　处理结果

5.5　本章小结

本章介绍了如何使用编译好的并行 OpenCV 库进行并行图像处理。5.1 节介绍了如何安装编译软件 CMake 和并行开发工具 TBB。5.2 节详细介绍了如何将 TBB、OpenCV、CUDA 混合编译到一起，生成支持 GPU 的并行 OpenCV 库。5.3 节介绍了如何搭建新编译好的并行 OpenCV 库。5.4 节使用并行 OpenCV 库进行图像处理，实现了反向算法、图像的加法和减法、图像的腐蚀和膨胀以及非局部均值算法。

参考文献

[1]　https://baike.baidu.com/item/cmake
[2]　高薪，胡月，杜威，等.腐蚀膨胀算法对灰度图像去噪的应用[J].北京印刷学院学报，2014(4)：63-65.